TEAMCENTER

Server Customization

Teamcenter 14.2

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit **www.siemens.com/plm**.

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

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

Contents

Getting started with server customization

Data-model-based customization

Teamcenter Services customization

ITK customization

How to develop an application

1. Getting started with server customization

Before you begin

Prerequisites	The following are required for both data-model-based framework customization and Integration Toolkit (ITK) customization:

- A tool to write code

    - For data model and services customization, install the BMIDE as described in *Configure Your Business Data Model in BMIDE*.

    - For ITK customization, use the C++ integrated development environment (IDE) of your choice.

- A compiler for compiling code

    Use a C++ compiler for all server-side customizations (ITK, services, and data-model-based customization).

    For supported compilers, including supported versions of Microsoft Visual Studio, see the Hardware and Software Certifications knowledge base article on Support Center.

- A test installation of a Teamcenter server and client to test custom code

    For Teamcenter installation instructions, see the server installation manuals, the *Teamcenter Installation on Windows* and the *Teamcenter Installation on Linux*. For client installation instructions, see *Rich Client Installation on Windows* and the *Rich Client Installation on Linux*.

Enable server customization	To *enable data-model-based customization* or services customization, install the Business Modeler IDE.
Configure server customization	To configure data-model-based customization or services customization, set up the coding environment in the **Business Modeler IDE**.

Methods for customizing the Teamcenter server

You can customize the Teamcenter server using the following methods:

- Business Modeler IDE

An integrated development environment (IDE) tool that allows you to customize the server using business object and property operations, extensions, and user exits.

- Teamcenter Services

A server-side customization method that uses standard service-oriented architecture (SOA) to provide the framework.

- Integration Toolkit (ITK)

An older, server-side customization method.

Application interfaces

There are two application interfaces you can use to integrate external applications with Teamcenter:

- Teamcenter Services

Teamcenter Services allow different applications to communicate with one another using services. Teamcenter Services use service-oriented architecture (SOA), which is appropriate for integrating separate applications with a loose linkage and network-friendly APIs that do not need to be modified with every Teamcenter version. This is the preferred method of integrating third-party and client applications. Future releases will extend this interface and provide tools so you can easily extend it.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

- Integration Toolkit (ITK)

Use the ITK interface when programming within the Teamcenter server (for example, extension points, workflow handlers, or low-level batch programs) and when constructing new interfaces for Teamcenter Services. It is a low-level interface that normally requires many more calls than the higher level interface (Teamcenter Services) to execute a given action, but it is powerful when you need to access the system in this way. For integrating external applications, Siemens Digital Industries Software recommends that you use Teamcenter Services. If you need functionality not exposed using Teamcenter Services, write new, high-level service methods using ITK and expose them as Teamcenter Services interfaces using the Teamcenter Services toolkit.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

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 conventions used in this guide

This guide 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
```

Basic server concepts

Teamcenter data model
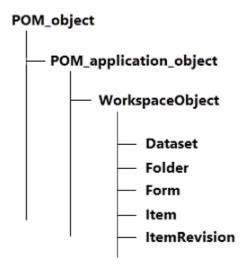
Teamcenter data model structure

To successfully perform customizations, you must understand the Teamcenter *data model*, which includes the collection of data modeling objects that represent real-world objects and their relationships.

The Teamcenter persistent object manager (POM) defines the data model architecture, or schema, using classes and business objects.

Element	Description	Appearance in BMIDE Advanced perspective
Classes	The persistent representations of the schema. Each class is mapped to a *primary business object* whose name is the same as the class name. In the Business Modeler IDE, classes can be seen in the **Advanced** perspective **Classes** view.	 **Classes view (POM schema)**
Business objects	The logical representations of the classes. Also known as *types*. In the Business Modeler IDE, business objects can be seen in • the **Advanced** perspective **Business Objects** view, • and in the **Standard** perspective **BMIDE** view.	 **Business object view (logical representation of the schema)**

Business objects represent the different kinds of objects you want to manage, such as items, datasets, forms, folders, and so on. The following figure shows these objects in an abbreviated view of the schema.
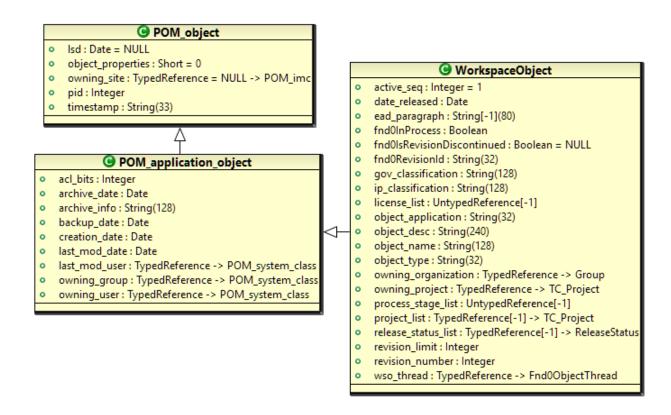
Abbreviated view of the schema showing object types

To see the information on a class or business object, in the Business Modeler IDE right-click the object and choose **Open in UML Editor**.

To see the inheritance relationships the object has to other objects, in the UML Editor right-click the object and choose **Show→Inheritance to Root**.

Example:

The following image is taken from the UML Editor and shows the inheritance of the **WorkspaceObject** class.

Inheritance of the WorkspaceObject class

> Tip:
>
> For more information about how to work with the UML Editor, see *Configure Your Business Data Model in BMIDE*.

Basic data model terms

Term	Definition
Class	A class is the definition of an object implemented in the Teamcenter data model. A class has a set of attributes, some of which are inherited from parent classes and some of which are defined directly for the class.
Business object	A business object is the specialization of a Teamcenter class based on properties and behavior.
Primary business object	A primary business object corresponds to each POM class (that is, the primary business object name is the same as the POM class name). Teamcenter automatically creates the primary business object when a new POM class is instantiated.
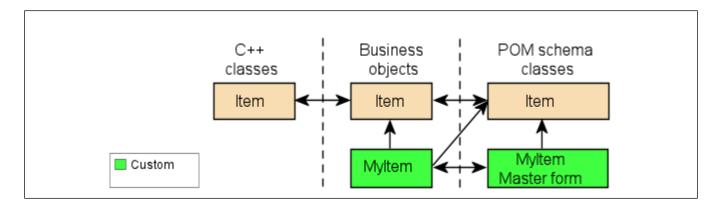
Term	Definition
Secondary business object	Secondary business objects are all business objects that belong to the same POM class. Secondary business objects inherit all the properties and behavior of the primary business object.
Attribute	An attribute is a persistent piece of information that characterizes all objects in the same class.
Property	A property is a piece of information that characterizes all instances of the same business object. At a minimum, all sub-business objects have properties that correspond to the attributes for the associated class. Properties can also represent other types of information such as relationships and run-time data.

Introduction to primary business objects

A *primary business object* corresponds to each POM class, and the primary business object name is the same as the POM class name. The primary business object uses its corresponding POM class to store its data. (Teamcenter automatically creates the primary business object when a new POM class is instantiated.)

However, a *secondary business object* uses its parent POM class as its storage class (see the following figure). Prior to Teamcenter 8, each POM class mapped to multiple business objects, and all business objects were secondary. In this scenario:

- Custom properties were added to a master form.

- ITK extensions in prevalidation, postvalidation, and validation actions and user exits were the only way to extend functionality.

- C++ classes were available only to Siemens Digital Industries Software developers and were not exposed. C++ classes mapped to one POM schema class and implemented the behavior for the primary business objects. Business logic for custom business objects (for example, **MyItem**) was added into the parent C++ class, or was written into separate ITK functions outside the C++ class hierarchy.
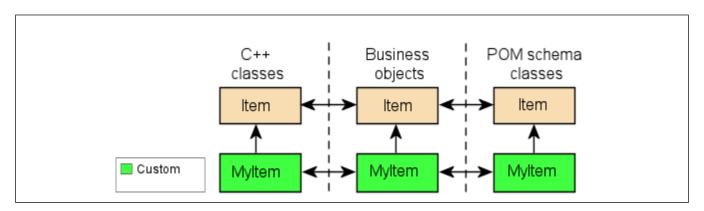
Secondary business objects (pre-Teamcenter 8)

After Teamcenter 8, the business object hierarchy is primarily one-to-one with the POM schema class hierarchy. In this scenario:

- Properties can be added directly to the custom business objects.

- There are still ITK extensions used for prevalidation, postvalidation, and validation actions.

- C++ operations are available to everyone. The C++ class hierarchy is one-to-one with the business object hierarchy.

- You can create new operations and services. You can put the business logic for custom business objects in the C++ class (for example, **MyItem**) and use the inheritance model to add onto the parent behavior or override custom behavior.

You can continue to use the master form to support legacy customizations or improve loading performance when using large amount of properties.

Primary business objects (post-Teamcenter 8)

> Note:
>
> Secondary business objects can be converted to primary business objects using the Business Modeler IDE by right-clicking the secondary business object in the **Business Objects** view of the **Advanced** perspective and choosing **Convert to primary**.
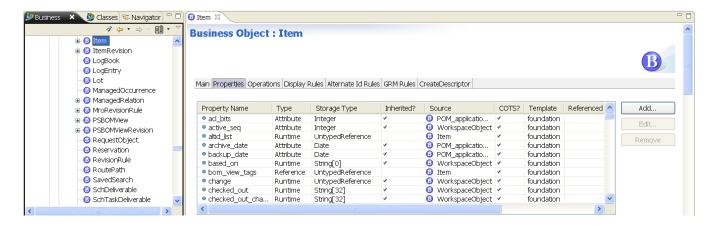
Introduction to properties

A *property* is a piece of information that characterizes all instances of the same business object. To see all the properties on a business object, in the Business Modeler IDE open the business object from either

- the **BMIDE** view in the **Standard** perspective

- or the **Business Objects** view in the **Advanced** perspective

and click the **Properties** tab.

Properties tab

Properties contain information such as name, number, description, and so on. A business object derives its persistent properties from the attributes on its persistent storage class. Business objects can also have additional properties such as run-time properties, compound properties, and relation properties.

You can add the following kinds of properties:

- Persistent

  Persistent properties are properties of business objects that remain constant on the object.

- Run-time

Run-time properties are derived each time the property is displayed. Their data is derived from one or more pieces of system information (for example, date or time) that are not stored in the Teamcenter database.

- Compound

  Compound properties are properties on business objects that can be displayed as properties of an object (the display object) although they are defined and reside on a different object (the source object).

- Relation

  Relation properties are properties that define the relationship between objects. For example, a dataset can be attached to an item revision with a specification, requirement, or reference relation, among many others.

After you add a property, you can change characteristics of the property by using property constants. For example, by default, when you add a persistent property is it read only. To make it writeable, you must change the **Modifiable** constant on the property from **Read** to **Write**.

Libraries

A *library* is a collection of files that includes programs, helper code, and data. Keep the following in mind when working with libraries:

- Create a library in the **Advanced** perspective of the Business Modeler IDE by opening the **Code Generation**→**Libraries** folders in the **Extensions** view, right-clicking the **Libraries** folder, and choosing **New Library**.

- After you write your C++ code, choose **Project**→**Build Project** in the Business Modeler IDE to build the libraries containing server code.

- If you have codeful customizations, you cannot deploy from the Business Modeler IDE for testing because codeful customizations have libraries. Instead, you must package your template so that libraries are placed into a *template-name*_**rtserver.zip** file, and then install the packaged template using Teamcenter Environment Manager (TEM).

- Ensure that libraries are placed in the correct directories:

  - Ensure that by default the **TC_LIBRARY** environment variable is set to *TC_ROOT***lib** on all platforms.

  - Place libraries for user exits, server exits, and custom exits (supplier custom hooks) in the directory specified by the **TC_USER_LIB** environment variable (preferred), the **TC_BIN** environment variable (for Windows), or the **TC_LIBRARY** environment variable (for Linux).

  - Place libraries for legacy operations in any directory as long as it is listed in the **BMF_CUSTOM_IMPLEMENTOR_PATH** preference.

- Place libraries for the C++ operations in the directory specified by the **TC_USER_LIB** environment variable (preferred) or the **TC_LIBRARY** environment variable (Windows and Linux).

> Note:
>
> For ITK, the **mem.h** header file moved from the **libitk** library to the **libbase_utils** library. The **mem.h** header file contains low-level methods related to memory allocation and release. These methods were exported from the **libitk** library and had circular dependencies with some application libraries that affected delay loading of libraries. To address this issue, a new low-level **libbase_utils** library is introduced. The functions in the **mem.h** file are now exported from this new library.
>
> The old header **mem.h** file still exists and includes the new **base_utils/Mem.h** header. Existing customization code should compile without any changes. However, as the functions are now exported from the **libbase_utils** library, the existing link dependency on the **libitk** library does not work.
>
> For any custom code using functions from the **mem.h** file, link dependency on the **libbase_utils** library must be added. For code where usage of the **libitk** library is limited to only functions from the **mem.h** file, the existing dependency on the **libitk** library can be removed in its entirety.

Install sample files

Sample files are provided to help you learn how to leverage Teamcenter technologies and extension points. Siemens Digital Industries Software provides guidance in the samples that is both useful and aligned to best practices, but samples are provided on an unsupported basis.

> Caution:
>
> Use samples at your own risk. Samples are subject to removal or change between releases.

1. Start Teamcenter Environment Manager (TEM).

2. In the **Maintenance** panel, select **Configuration Manager** and click **Next**.

3. In the **Configuration Maintenance** panel, select **Perform maintenance on an existing configuration** and click **Next**.

4. In the **Configuration** panel, select the configuration from which the corporate server was installed. Click **Next**.

5. In the **Feature Maintenance** panel, under the **Teamcenter** section, select **Add/Remove Features**. Click **Next**.

6. In the **Features** panel, under **Server Enhancements**, select **Sample Files**.

7. Click **Next**.

8. In the **Confirm Selections** panel, click **Next**.

The sample files are placed at the following location:

server-install-location`\sample`

Writing warnings or errors to logs

Use the **TC_write_syslog** API to write warnings to the syslog, and use the **TC_report_serious_error** API to write errors to the syslog. To use these functions, include the **tc.h** file.

• **TC_write_syslog** API

Use this function to write warnings or information level statements to the Teamcenter syslog file, for example:

```
extern TC_API void TC_write_syslog (
    const char*     control_string,    /**< (I) */
    ...                                 /**< (I) */
    );
```

Depending on the **control_string** string, the function may expect a sequence of additional arguments, each containing one value to be inserted for each **%-tag** specified in the **control_string** parameter, if any. There should be the same number of these arguments as the number of **%-tag** that expect a value.

Parameters	Description
control_string	Message to be written to **syslog** file. It can contain embedded format tags that are substituted by the values specified in subsequent arguments and formatted as requested. The number of arguments following the **control_string**' parameters should be at least as much as the number of format tags.

• **TC_report_serious_error** API

Use this function to write error messages to Teamcenter **syslog** file, for example:

```
extern TC_API void TC_report_serious_error (
    const char*     file_name,         /**< (I) */
    int             line_number,       /**< (I) */
    const char*     control_string,    /**< (I) */
    ...                                 /**< (I) */
    );
```

Parameters	Description
file_name	The name of the file in which the error occurred.
line_number	The line number at which the error occurred
control_string	Message to be written to the syslog file. It can contain embedded format tags that are substituted by the values specified in subsequent arguments and formatted as requested. The number of arguments following the **control_string** parameters should be at least as much as the number of format tags.

Depending on the **control_string** string, the function may expect a sequence of additional arguments, each containing one value to be inserted for each **%-tag** specified in the **control_string** parameter, if any. There should be the same number of these arguments as the number of **%-tag** that expect a value.

2. Data-model-based customization

Introduction to data-model-based customization

Data-model-based customization means making changes directly to data model objects, such as business objects. The Teamcenter data model framework supports a single, coherent mechanism for developing new functionality in C++ by defining business logic on business objects in the **Impl** class. This exposes the business logic API in an object-oriented way. Coding customization typically involves adding a new operation to a business object or overriding an implementation of an existing operation on a business object. Use the Business Modeler IDE to perform this work.

Operations are Teamcenter functions, such as create, checkin, checkout, and so on. Operations can be placed on custom business objects and properties. Operations on a business object provide the interfaces for business logic. The implementation of the business logic is done by functions or methods on the implementation class corresponding to the business object.

Customization methods in the Business Modeler IDE

If your organization wants to extend the functionality of Teamcenter by writing C++ code, you can use the Business Modeler IDE to write the code.

There are several ways to do customization:

- **Data-model-based customization**

  Allows addition of custom C++ operations to business objects and the overriding of existing operations on business objects. Customization typically involves this adding of new operations or the overriding of existing operations.

- **Teamcenter Services customization**

  Allows custom service-oriented architecture (SOA) service operations to be written to interact with the Teamcenter rich client.

  Teamcenter Services insulate the client tier from changes to server behavior and the low level data model objects. These are less granular services that improve the performance of client communication in a WAN environment.

- Extensions customization

  Allows you to write a custom function or method for Teamcenter in C or C++ and attach the rules to predefined hook points in Teamcenter (preconditions, preactions, and postactions). Also, existing operations can be extended to these hook points.

Set up a Business Modeler IDE project for coding

You set up coding information for a project at three different points as needed:

- During the Business Modeler IDE installation or configuration

- When a new project is created

- For an existing project, when coding information is added or modified

Specify coding information when you install or configure the Business Modeler IDE

- When you install or configure the Business Modeler IDE, set up the JDK location and a compiler application.

> Example:
>
> The **bmide.bat** file starts the Business Modeler IDE. Set the location of the JDK in the *install-location***\bmide\client\bmide.bat** file:
>
> ```
> set JRE_HOME= ... \Java\jrex
> set JAVA_HOME= ... \Java\jdkx.x.x
> set JDK_HOME= ... \Java\jdkx.x.x
> ```
>
> On Windows when Microsoft Visual Studio is used for compiling, a call to the **vcvarsall.bat** file (along with any necessary arguments) in the **bmide.bat** file is required. The call should be before the **PATH** statement, such as:
>
> **call "C:\ ... \vcvarsall.bat" amd64**
> ```
> set PATH= ...
> ```

Specify coding information when you create a new project

1. In the Business Modeler IDE, choose **File**→**New**→**Project**.

2. In the **New Project** dialog box, choose **Business Modeler IDE**→**New Business Modeler IDE Template Project**.

3. Enter information in the pages of the **New Business Modeler IDE Template Project** dialog box.

   On the **Code Generation Information** page, you can set up an optional location for generated source files.
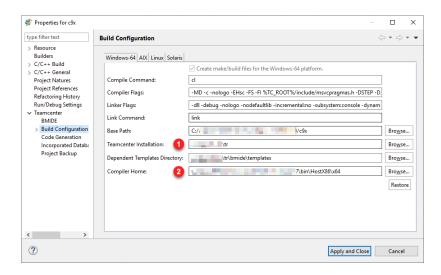
You can also set your deprecation policy.

On the **Build Configuration Information** page, provide the:

a.   Teamcenter root location

b.   Compiler home location

Add or modify coding information for an existing project
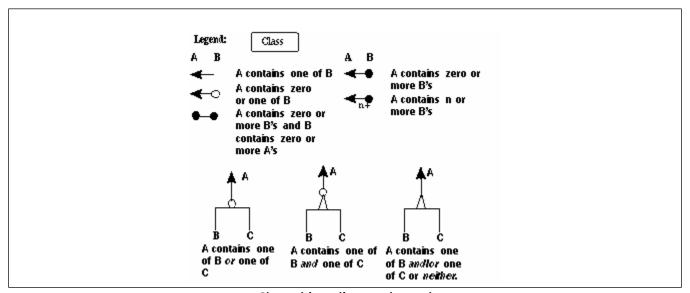
- To set up code generation for an existing project, modify the project's properties. Right-click the project, choose **Properties**, and select **Teamcenter→Code Generation**.

Teamcenter object model

Legend for object diagrams

The Teamcenter object model contains the classes used to manage data. The model is described using a set of two basic relationship diagrams. One set of relationships show the class hierarchy (taxonomy) and the other shows the associations that exist between interacting objects in the system.

Following is the legend for the class object diagrams.
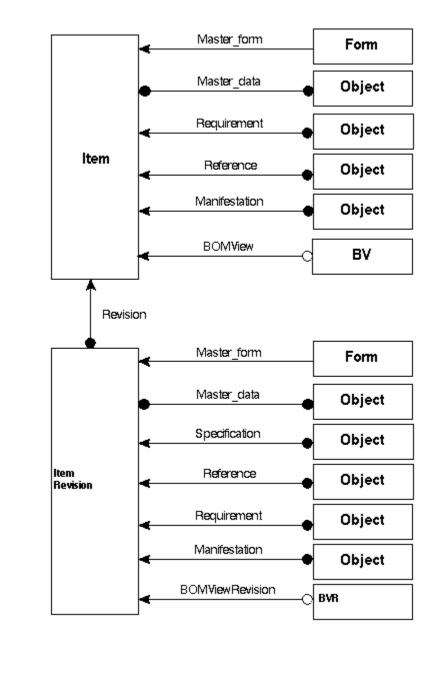
Class object diagram legend

Item and item revision model

Sample item and item revision model

The following figure shows an illustration of an item/item revision model.

Item/item revision model

Item

In Teamcenter, items are the fundamental objects used to manage information. Items provide an identification for physical or conceptual entities about which an organization maintains information and audit/change control. Typical uses of items are parts, documents, and equipment. In an engineering/product development environment, parts are identified as items. An item has the following basic information:

- **ID**

A unique identifier for an item.

- **Name**

  A user-defined name to describe the item.

- **Item type**

  A classification of items that allow different kinds of items to be treated separately. For example, an item can be used to manage parts and documents. If you implement two item types, you can enforce rules based on the type of items that are being manipulated.

  > Note:
  >
  > The system, as delivered, provides a generic business object called **Item** along with several child business objects which all act as different item types. If you need additional kinds of items, the system administrator can create children of the **Item** business object at your site.

- **Description**

  A text field of up to 240 characters used to describe the item.

Item revision

Item revisions are used to reflect modifications to an item. The original item revision is retained for historical purposes.

> Note:
>
> Each customer site defines its own procedures to determine how and when a new item revision should be created.

An item revision has the following basic information:

- **ID**

  A unique identifier for an item.

- **Revision**

  A unique identifier for an item revision.

- **Name**

  A user-defined name to describe the item revision.

Relations

Organizations produce several documents that describe, are used by, or in some way relate to an item. For the data to be understood, it has to be associated in a meaningful way to the item or item revision. Teamcenter associates data to items and item revisions using relations.

Relations describe how the data is associated to an item and item revision. A dataset containing the CAD model, which defines an item revision, is a specification of the item revision. Similarly, a standards document describing the fit, form, and function of a particular drawing is a requirement for the item.

An item or item revision can be related to several other objects, including datasets, forms, folders, and other items and item revisions. Each object associated to the item represents various aspects of that item. A dataset containing stress test information could be added by the engineer, while a form containing size and weight information could be added by the specification engineer.

Teamcenter provides a basic set of relations. The relations between the object and the item and item revision determines the rules enforced.

> Note:
>
> Additional relations and their associated rules are defined by your system administrator.

- **IMAN_master_form**

  The master form relation associates a form to an item or item revision. This form is a collection of attributes that describe the item or item revision. Rules for this relation are as follows:

  - An item or item revision must have write access to add or remove a master form relation to a form.

  - Only a form can have a master form relation to an item or item revision.

  - An item can have only one master form. An item revision can have only one master form.

  - A form can have a master form relation to only one item or item revision.

- **IMAN_requirement**

  The requirement relation associates data to an item or item revision, such as standards, which must be adhered to in a drawing, or technical requirements for a part. Rules for this relation are as follows:

  - An item or item revision must have write access to add or remove a requirement relation to an object.

  - A form can have a requirement relation to an item or item revision.

  - Only version 0 (zero) of a dataset can have a requirement relation to an item or item revision.

- A folder, envelope, BOM view, or BOM view revision cannot have a requirement relation to an item or item revision.

- An item cannot have a requirement relation to another item or item revision.

- An item revision can have a requirement relation to another item or item revision.

- An item revision cannot have a requirement relation to another item revision if they are both revisions of the same item.

- **IMAN_manifestation**

  The manifestation relation associates other related data to an item or item revision. This data may, however, be necessary for information, such as analysis of the competing ideas from which a part was originally conceived. The manifestation relation also associates data to the item revision that contains data derived from the specification data (such as tool paths). Rules for this relation are as follows:

  - An item or item revision does *not* need to have write access to add or remove a manifestation relation to an object. A manifestation relation can be added or removed from a released item or item revision.

  - A form can have a manifestation relation to an item or item revision.

  - Only version 0 (zero) of a dataset can have a manifestation relation to an item or item revision.

  - A folder, envelope, BOM view, or BOM view revision cannot have a manifestation relation to an item or item revision.

  - An item cannot have a manifestation relation to another item or item revision.

  - An item revision can have a manifestation relation to another item or item revision.

  - An item revision cannot have a manifestation relation to another item revision if they are both revisions of the same item.

- **IMAN_specification**

  The specification relation associates data which defines the item revision. Examples are CAD models for parts or word processor files for documents. Rules for this relation are as follows:

  - An item revision must have write access to add or remove a specification relation to an object.

  - A form can have a specification relation to an item or item revision.

  - Only version 0 (zero) of a dataset can have a specification relation to an item or item revision.

- A folder, envelope, BOM view, or BOM view revision cannot have a specification relation to an item or item revision.

- An item cannot have a specification relation to another item or item revision.

- An item revision can have a specification relation to another item or item revision.

- An item revision cannot have a specification relation to another item revision if they are both revisions of the same item.

- **IMAN_reference**

  The reference relation associates any data to an item or item revision. Rules for this relation are as follows:

  - An item or item revision does not need to have write access to add or remove a reference relation to an object. A reference relation can be added or removed from a released item or item revision.

  - Any object can have a reference relation to an item or item revision.

  - Only version 0 (zero) of a dataset can have a reference relation to an item or item revision.

  - A reference object cannot reference itself.

- **IMAN_revision**

  The revision relation associates item revisions to the appropriate item. Rules for this relation are as follows:

  - An item must have write access to add or remove an item revision from the item.

  - Only an item revision can have a revision relation to an item.

  - The revision relation in an item is established by creating a new revision of an item using the provided functionality. An item revision cannot be cut or pasted into an item.

- **BOMView**

  The **BOMView** relation associates a product structure to an item. Rules for this relation are as follows:

  - An item can have many **BOMView** relations.

  - **BOMView** relations can represent different versions of a product structure. For example, one can be the design structure and another can be the manufacturing structure.

- **BOMView Revision**

The **BOMView Revision** relation associates a product structure revision to an item revision. Rules for this relation are as follows:

- An item revision can have many **BOMView Revision** relations.

- **BOMView Revision** relations can represent different versions of a product structure. For example, one can be the design structure and another can be the manufacturing structure.

Modifying an item/item revision

Teamcenter is designed to organize the data created by other applications in a logical manner. When an item is first created, an item ID is reserved. It is only by adding data to an item, however, that the item becomes meaningful.

Datasets are used to store information created by other applications. These datasets can be associated to an item and item revision.

In Teamcenter, both item revision and dataset versions are intended to represent modifications to information. Item revisions represent the item at significant milestones. Dataset versions represent day to day changes. An item has revisions; a dataset has versions.

There are two ways to modify an item and item revision. One method is to simply add or remove relations from the item and item revision by using cut and paste. Another method is to modify the contents of an object that is currently related to the item and item revision.

Item business object
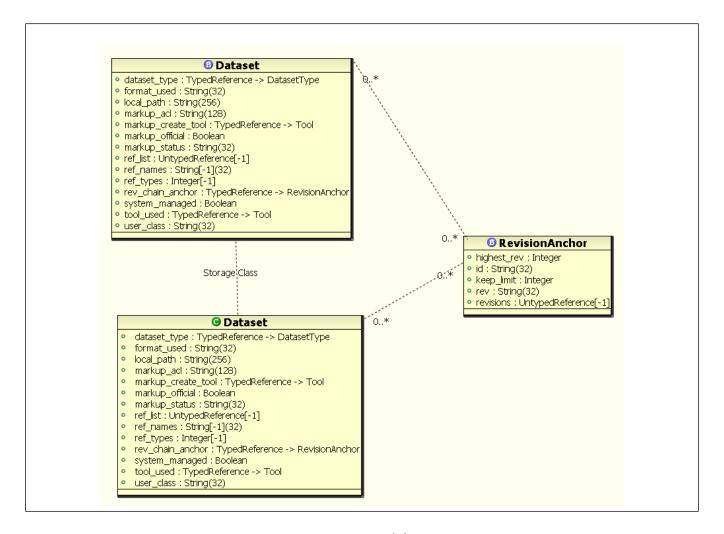
Business objects are used to specialize the behavior of Teamcenter objects. Datasets implement the most extensive use of business objects. The **Dataset** business object is used to categorize datasets.

Examples of dataset business objects are a document, an NX part, and a spreadsheet. When a document **Dataset** business object is opened, the word processor application associated to this **Dataset** business object is opened. When an NX part **Dataset** business object is opened, the NX application associated to this **Dataset** business object is opened. For each **Dataset** business object, the available operations (for example print and open) are defined, as well as the specific behavior of each operation.

Item business objects extend the ability to specialize the behavior of items. You may need to distinguish between a part used in production and equipment used to produce that part. Both may need to be an item, yet the information and business rules surrounding these items are different.

Dataset model

A dataset contains a series of dataset objects, which reference the **ImanFile** objects that hold the data, and a **RevisionAnchor** object, which holds the list of dataset versions and the dataset ID, if one exists. A dataset may also reference **Form** objects. The following figure from the UML (Unified Modeling Language) editor in the Business Modeler IDE shows the **RevisionAnchor** object that creates a new logical dataset.

Dataset model

Dataset IDs are optional, but they must be unique. The unique ID enforcement in datasets is performed at the application level. Each time the **New→Dataset** menu command is used to create a dataset, the **Save As** and **Import** commands are used, or the dataset is edited using the **Properties** command, the validation function is called to ensure the dataset ID (**id**) and revision (**rev**) combination is unique within that specified dataset business object.

Also, for each dataset business object, if the dataset ID is created using the **Assign** button, a separate counter generates the next unique ID that is available for a standalone dataset. This counter ensures **id** and **rev** are unique within the same dataset business object.

When you create a dataset, two revisions are created: revision 0 and revision 1. Revision 0 always points to the latest revision, so it initially points to revision 1. If you create revision 2, then revision 0 points to that, and so on. In your programming, if you want to point to the latest revision, point to revision 0.

ITK functions use this convention with revisions as follows:

• **AE_ask_dataset**

Given any dataset, returns revision 0.

- **AE_ask_dataset_first_rev**

  Given any dataset, returns the oldest revision.

- **AE_ask_dataset_latest_rev**

  Given any dataset, returns the most recent revision (not the revision 0 copy).

- **AE_ask_dataset_next_rev**

  Given any dataset, returns the next revision in the sequence.

- **AE_ask_dataset_prev_rev**

  Given any dataset, returns the previous revision in the sequence.

- **AE_find_dataset ("name", &dataset)**

  Finds revision 0 of the dataset.

- **AOM_refresh (dataset, true)**

  Loads the given dataset version for modification.

- **AOM_save (dataset)**

  Creates a new, later version of the dataset and saves a copy to revision 0.

- **AOM_delete**

  For the given dataset, deletes all revisions and the associated revision anchor.

All revisions of a dataset have the same owning user and group and the same protection; updating any of those on any one revisions updates all other revisions.
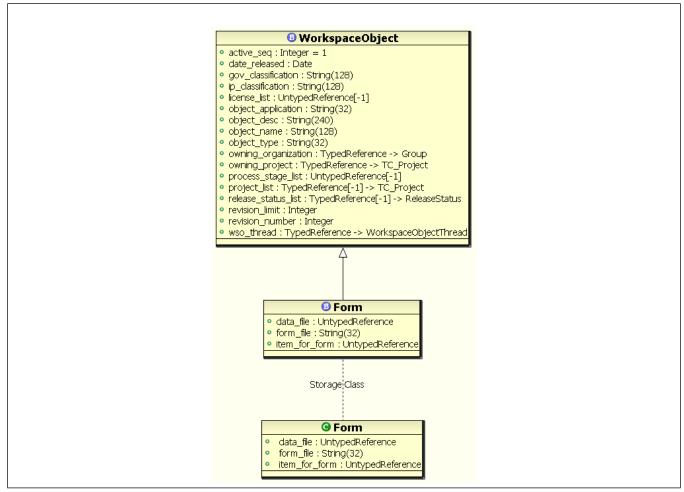
> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Form model

Forms provide the ability to store customer defined attributes. Teamcenter provides standard forms. Most of the forms used at a customer site, however, are modified to capture the information unique to its business.
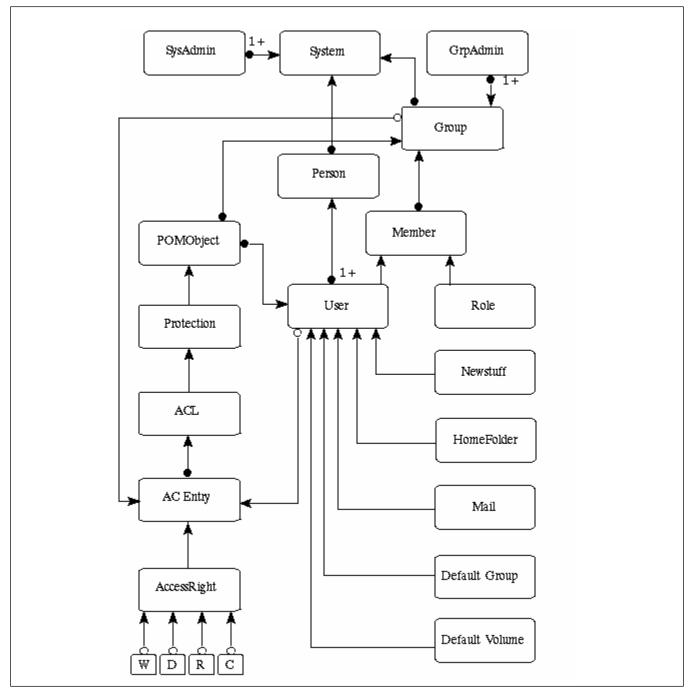
Forms are stored as POM objects. Attributes in the forms can be set and retrieved using **Form** ITK functions and the POM query ITK functions. When a form is stored, the database schema must be updated to recognize the new POM object. The following figure shows a diagram of the form model using the UML (Unified Modeling Language) editor in the Business Modeler IDE.

Form model

System administration model
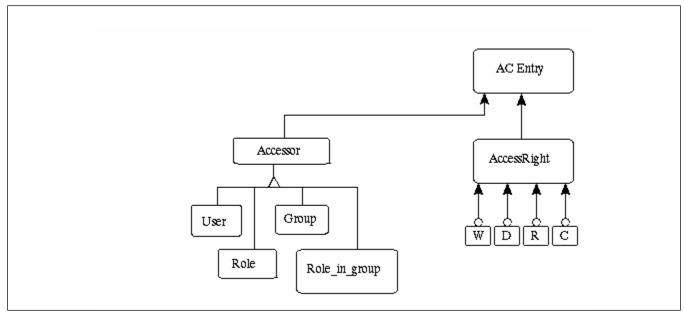
The **POM_application_object** class and all its subclasses have owners and protection that can be altered by the system administrator. The following figure shows a diagram of the system administration model.

System administration model

Access control model

Access control list (ACL) entries consist of a reference to a group and/or a user and the access rights grant to such a subject. A subject is an object that can be granted access rights in Teamcenter. The following figure shows a diagram of the access control model.

Access control model

Application encapsulation model

The application encapsulation data model is designed to enable Teamcenter to maintain relations between data and the applications that can present it to the end user.

The model provides a means of declaring an application to Teamcenter by creating instances of the **Tool** class. It also provides a means of determining which tool to use for the creation and manipulation of various pieces of data. This is done by providing a **DatasetType** object class for you to use when specifying the tools that can create and manipulate data.

A tool can be a shell. A *shell* is an executable image written to execute other legacy images. The shell uses the ITK to extract data from Teamcenter and update Teamcenter with data created by the execution of legacy systems. A shell can be developed that manages all applications at a site; it can be started from the command line. This shell can present a form to the user to fill out before executing the legacy system so the shell can make any records that the site may need.

A shell can also be developed to copy data from Teamcenter into a temporary or working directory from which a legacy (non-integrated) application is run. This allows some support for applications that always require interactive input. If the shell moves required files into the working directory, and the application reads files from the current directory if only file names are specified, it would be very difficult to put procedures in place that require the user to use only file names when using such applications. If this is done, a shell can prepare data for an application, record the state of the directory and use that information to determine what happened after the application is terminated. In this type of example, the input/output parameters of the application declaration in Teamcenter can allow this shell to ignore some scratch files that may be left in the directory by the executing application.

If you start from scratch, you can develop an integrated application that does not require a shell.

The **Dataset** class is the workspace object that an end user sees in their workspace. The shell or integrated application associates one or more other pieces of data to the dataset. More often than not, this means associating files for existing applications.

Some applications require several files in order to run. When this is the case, a named reference can be created for each file. The name of the reference is often the extension of the file but does not have to be. A CAD system may have a file in which it stores geometric data with a **.gmd** extension, finite element modeling data in a file with a **.fem** extension, and drawing data in a file with a **.drw** extension.

For such a CAD system, you could find references named **geometry**, **fem**, and **drawing**, or **gmd**/**fem**/**drw** as the shell chooses. If you use the extensions, users do not have to change the reference names during import, since by default the import assumes the reference names correspond to the file extensions.

The model shows that the named reference can refer to an **ImanFile** object or other descendant class of the POM object. This means a shell can be implemented that does not use the **ImanFile** object to store file data, but instead uses its own file object to store such information. However, you should do this rarely since there is a loss of integrity that is provided by the **ImanFile** object. However, the flexibility it provides may be necessary in some cases.

The **RevisionAnchor** object is provided to support versioning of datasets. Whenever a dataset is created, a **RevisionAnchor** object is also created. This object maintains a list of working revisions of the dataset. However, it is up to the shell/integrated tool to use this facility. To use it, you must make revision copies and then make changes to these copies.

Class hierarchy

The information in this section describes the hierarchy of classes (taxonomy). Understanding this hierarchy, or returning to it, helps with understanding the rest of the Teamcenter model description, which shows relationships and gives a more detailed description of individual classes.
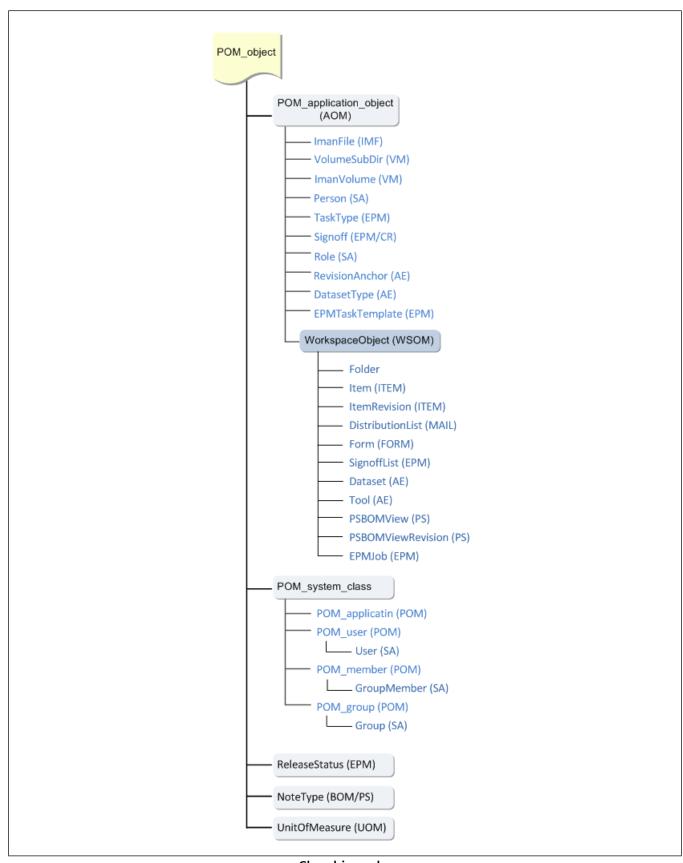
Class objects have a meaningful hierarchy that relates to the Teamcenter layered software architecture previously discussed. It is presented in an indented outline form.

The following legend defines the mnemonic descriptions shown in parentheses in the figure.

Mnemonic	Class
AE	Application Encapsulation
AOM	Application Object Module
BOM	Bill Of Materials
CR	Cascade Release
EPM	Enterprise Process Modeling
FL	Folder Manager

Mnemonic	Class
FORM	Forms
IMF	ImanFile
ITEM	Item
MAIL	Teamcenter Mail
POM	Persistent Object Manager
PS	Product Structure
SA	System Administration
UOM	Unit of Measure
VM	Teamcenter Volumes
WSOM	Workspace Object

The following figure shows the more common classes in the hierarchy.

Class hierarchy

Mapping business objects and classes

Starting in Teamcenter 8, secondary business objects are mapped to their own subclasses and attributes are stored directly in the subclass for the entire hierarchy. In previous versions, secondary business objects were mapped to the parent classes of their primary business objects and secondary business object attributes were stored on the master form. If you upgrade from a previous version of Teamcenter or Teamcenter engineering process management to Teamcenter 8 or a later version, the metamodel migration upgrade converts the secondary business objects to their own subclasses.

> Warning:
>
> You may lose data if you use Multi-Site Collaboration with a system that has both pre-Teamcenter 8 and Teamcenter 8 or later sites.
>
> • If you transfer data from a Teamcenter 8 or later site to a pre-Teamcenter 8 site, any attributes you added to the subclasses in Teamcenter 8 or later versions are dropped at the pre-Teamcenter 8 site.
>
> • Similarly, if you transfer data from a pre-Teamcenter 8 site to a Teamcenter 8 or later site, any attributes you added to the master form for the secondary business objects in the pre-Teamcenter 8 version are dropped at the Teamcenter 8 or later site.
>
> To avoid data loss when transferring data from a Teamcenter 8 or later site to a pre-Teamcenter 8 site, create new attributes on the master form in Teamcenter 8 or later versions.

Object-oriented data

What is an object?

An object is a data structure. It is basically the same as an entity or an instance of a class. A class defines a type of object, its attributes, and methods that operate on it.

For example, **folder** is a class in Teamcenter. It is defined to have a name, a description, and a list of entries. These are the attributes of the **folder** class. When a folder is instantiated, an actual folder object is created. It has attributes as described here. There are also methods or functions defined for the folder class, such as **Insert** and **Remove**.

Attributes in Teamcenter can be integer, float, boolean, string, date, tag, or arrays of any of these types. A tag is a unique identifier of a reference (typed, untyped, or external) or a relation (typed or untyped).

The attributes in Teamcenter can have some of following characteristics:

• Unique

   There can be no duplicate values of the attribute in all of the instances of the class.

- Protected

  The attribute can only be modified by the application that created the object.

- NULL allowed

  If this characteristic is not true, then the object cannot be saved until the attribute is set.

- Upper bound

  The highest value that the attribute can have.

- Lower bound

  The lowest value that the attribute can have.

- Default value

  The attribute may have an initial value.

Inheritance

A subclass inherits both attributes and methods from superclasses.

Multiple inheritance means that a class has two superclasses. That is, it inherits the attributes and methods from two different classes. There are no instances of multiple inheritance in Teamcenter.

A superclass may be a generalization of many similar classes. For example, the **WorkspaceObject** superclass is a generalization for all of the classes whose instances may be seen in the workspace (for example, in a folder). All of the attributes and methods that are common to the **Dataset**, **Folder**, and other classes are defined once in the **WorkspaceObject** superclass, rather than being defined many times. Usually, generalizations are abstract or noninstantiable. This means that the class exists for conceptual or convenience reasons, but you cannot actually create an instance of one.

A subclass may be a specialization of another class. For example, the **Envelope** subclass is a specialization of the **Folder** class. In this case, a new class is being defined that is like another one except for some additional attributes and/or some specialized behavior such as a specialized method.

Object-oriented language

An object-oriented language is a programming language that has built in key words or constructs that either force or facilitate object oriented programming. For example, C++ is one, because it supports classes, objects, inheritance, and polymorphism. The C language is not.

However, you can do object-oriented programming with a nonobject-oriented language; it is just harder. You can certainly have an object model with inheritance of attributes and methods without using an object oriented language. This is what is presented through ITK.

Internally, much of Teamcenter is written in C++. This provides an easy way of handling polymorphism. You can take advantage of this in the ITK as well. If you want to execute a method on an object, you do not need to call a function specific to that action-class pair. You may call any function with that action up the tree of superclasses for the selected object. For example, if you want to ask the name of an envelope, you do not need to call the **EMAIL_ask_envelope_name** function (in fact, that function does not exist). You can call the **WSOM_ask_name** function two levels up in the hierarchy instead. That function realizes that it was actually passed an envelope and invokes the envelope's method to get the name.

Some of the important concepts of object-oriented programming are:

- Encapsulation

  Combining data types with valid operations to form a class. A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain type.

- Inheritance

  The ability to inherit class definitions and make modifications that specialize the class, thus creating a new class.

- Polymorphism

  Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

Data-model-based customizations

Introduction to data-model-based customizations

The Teamcenter data model framework supports a single, coherent mechanism for developing new functionality by defining business logic on business objects in the **Impl** class. This exposes the business logic API in an object-oriented way (through interfaces).

Customization typically involves adding a new operation to a business object or overriding an implementation of an existing operation on a business object.

Operations are Teamcenter functions, such as create, checkin, checkout, and so on. Operations can be placed on business objects and properties. To see these operations, right-click a business object, choose **Open**, and in the resulting editor, click the **Operations** tab.

Operations on a business object provide the interfaces for business logic. The implementation of the business logic is done by functions or methods on the implementation class corresponding to the business object. There are certain restrictions on operations. You cannot modify a COTS operation (that is, noncustom operation). You cannot modify or delete a released operation; you can only deprecate it.

Coding process

Following is the general process for writing data-model-based customization code in the Business Modeler IDE. The **Code Generation** folder under the **Extensions** folder is where you do most of this work.

1. **Set up the coding environment**.

   Set up code generation, and create the release, library, and data types you want to use.

2. **Create an operation** or **override an operation**.

3. **Generate boilerplate code**.

   Generate boilerplate C++ code files in which you can write your code implementation for the operation.

4. **Write the implementation code**.

   Write your implementation of the operation in the generated template files.

5. **Build server code**.

   Build the libraries containing server code.

6. Package and install.

   Package your changes into a template and install the template to a server. (You cannot use live update to distribute customizations to a server.)

   > Note:
   >
   > The Business Modeler IDE packages the built C++ library as part of its template packaging. The complete solution that includes the data model and the C++ run-time libraries are packaged together.
   >
   > The Teamcenter *C++ API Reference* available on Support Center documents APIs in C++ signatures.

Set up the coding environment

Before you use the Business Modeler IDE to write C++ code, you must set up your coding environment.

1. **Set up code generation**.

   Define where the generated code files are placed.

2. **Create a release**.

   Define the software release that the generated code is used in.

3. **Create a library**.

   Make a set of library files that will hold the generated code.

4. If needed, **create external data types**.
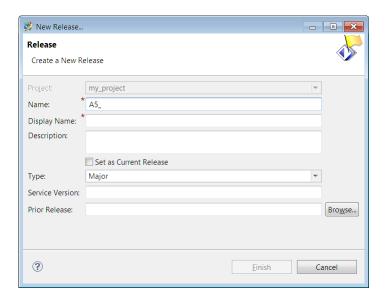
   Create external data types to represent new kinds of data.

Create a release

Releases are distributions of software products. By default, Teamcenter releases are listed under the **Extensions\Code Generation\Releases** folder so that you can create code against these releases. You can specify the release against which code is created by right-clicking a release and choosing **Organize→Set as active release**. The code you create after setting the release is then assigned to that release. When you create some code objects, such as operations or services, you must specify a release.

1. Choose one of these methods:

   • On the menu bar, choose **BMIDE→New Model Element**, type **Release** in the **Wizards** box, and click **Next**.

   • In the **Extensions** folder, open the **Code Generation** folder, right-click the **Releases** folder, and choose **New Release**.

   The New Release wizard runs.

2. Perform the following steps in the **Release** dialog box:

   a. The **Project** box displays the project to which this new release is added.

   b. In the **Name** box, type the name you want to assign to the new release in the database.

   When you name a new data model object, a prefix from the template is automatically affixed to the name to designate the object as belonging to your organization, for example, **A4_**.

   c. In the **Display Name** box, type the name of the release as it will display in the Business Modeler IDE.

   d. In the **Description** box, type a description of the new release.

   e. Select **Set as Current Release** to select this new release as the active one to use for all data model creation processes.

   You can also set a release as active by right-clicking the release and choosing **Organize**→**Set as active Release**. A green arrow in the release symbol indicates it is the active release.

   f. Click the arrow in the **Type** box to choose whether this release is considered a major release, a minor release (which is dependent on a major release), or a patch (to a minor or major release).

   g. In the **Service Version** box, type the version of the services you plan to create in the release using format _YYYY_MM_, for example, **_2010_06**. You _must_ fill in this box if you plan to create services for this release.

   h. Click the **Browse** button to the right of the **Prior Release** box to choose the previous release. If you are creating a minor release, choose the major release that this minor release follows. If you are creating patch, choose either the major or minor release that the patch is applied to.

      i.   Click **Finish**.

           The new release appears under the **Releases** folder.

3.   To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.
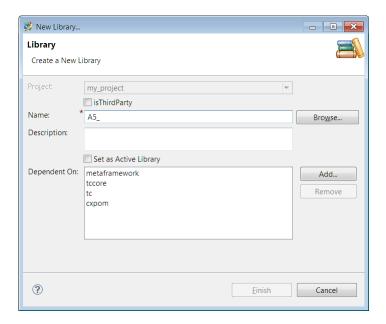
Create a library

A *library* is a collection of files that includes programs, helper code, and data. Open the **Extensions \Code Generation\Libraries** folders. By default, Teamcenter libraries are listed under the **Libraries** folder. You can create your own library that is dependent on Teamcenter libraries.

You can specify the library against which code is created by right-clicking a library and choosing **Organize→Set as active library**. The code you create after setting the library is then assigned to that library. When you create some code objects, such as business object or property operations, you must specify a library.

1.   Set the release the library is to be used for. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2.   Choose one of these methods:

- On the menu bar, choose **BMIDE→New Model Element**, type **Library** in the **Wizards** box, and click **Next**.

- In the **Extensions** folder, open the **Code Generation** folder, right-click the **Library** folder, and choose **New Library**.

    The New Library wizard runs.

3. Perform the following steps in the **Library** dialog box:

   a. The **Project** box displays the project to which this new library is added.

   b. Select the **isThirdParty** check box if the library is built outside of the Business Modeler IDE or the library is provided by a third-party vendor.

   c. In the **Name** box, type the name you want to assign to the new library, or if you selected the **isThirdParty** check box, click the **Browse** button to specify the third-party library file.

   d. In the **Description** box, type a description of the new library.

   e. Select the **Set as Active Library** check box to select this new library as the one to use for all data model creation processes.

       You can also set a library as active by right-clicking the library in the **Libraries** folders and choosing **Organize→Set as active Library**. A green arrow in the library symbol indicates it is the active library.

   f. Click the **Add** button to the right of the **Dependent On** pane to select the libraries this new library is dependent on. This box is disabled if you are using a third-party library file.

   g. Click **Finish**.

       The new library appears under the **Library** folder.

4. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 🖫 on the main toolbar.

Data types

Introduction to data types

In software programming, a *data type* is a definition for a specific type of data and the operations that can be performed on that data. For example, if some code has the **char** data type applied to it, it is character text and can contain only characters. Or if some code has the **int** data type applied, it is integer type code and can contain only numbers. You use data types as the return type when you create business object operations. To work with data types, open the **Extensions\Code Generation\Data Types** folders.

You can work with the following kinds of data types:

- **External data type**

  Custom data types that you can import into Teamcenter, such as date.

- **Primitive data type**

  Standard data types such as Boolean, character, double, float, and integer.

- **Template data type**
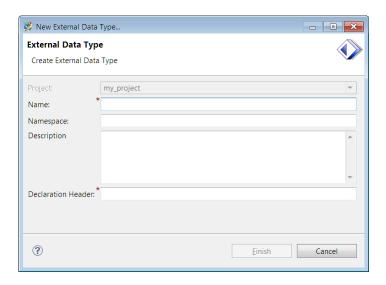
  Data types that are open-ended and allow generic programming.

Add an external data type

External data types are standard data types, as well as custom defined data types that can be imported into the Business Modeler IDE. You can use external data types as the return type when you **create business object operations**. To access the external data types, open the **Extensions\Code Generation \Data Types\External Data Type** folders.

1. Choose one of these methods:

   - On the menu bar, choose **BMIDE→New Model Element**, type **External Data Type** in the **Wizards** box, and click **Next**.

   - Open the **Extensions\Code Generation\Data Types** folders, right-click the **External Data Type** folder, and choose **New External Data Type**.

   The New External Data Type wizard runs.

2. Perform the following steps in the **External Data Type** dialog box:

   a. The **Project** box displays the project to which this new data type is added.

   b. In the **Name** box, type the name you want to assign to the new data type.

   c. In the **Namespace** box, type a namespace for the type. Namespaces allow for grouping code under a name to prevent name collisions.

   d. In the **Description** box, type a description of the new external data type.

   e. In the **Declaration Header** box, type the header file declaring the external data type. The header should provide how the header file should be included in code, for example:

      path/custom-data-type.**hxx**

   f. Click **Finish**.

      The new data type appears under the **External Data Type** folder.

3. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

4. Deploy your changes to the test server. Choose **BMIDE→Deploy Template** on the menu bar, or select the project and click the **Deploy Template** button ⬚ on the main toolbar.

The new data type is now available for use by operations.

External data types reference

External data types are standard data types, as well as custom defined data types that can be imported into the Business Modeler IDE, such as date. You can use an external data type as the parameter type when you **create a business object operation** or **service operation**.

To access the external data types, open the **Extensions\Code Generation\Data Types\External Data Type** folders. Following are the standard external data types:

- **date_t**

  Structure to represent a date.

- **std::string**

  String from the standard namespace.

- **std::vector<bool>**

  Boolean (true or false) vector.

- **std::vector<char>**

  Character vector.

- **std::vector<date_t>**

  Date vector.

- **std::vector<double>**

  Double vector.

- **std::vector<int>**

  Integer (number) vector.

- **std::vector<long>**

  Long string vector.

- **std::vector<PropertyDescriptor>**

  Property descriptor vector.

- **std::vector<std::string>**

String vector.

- **std::vector<tag_t>**

  Tag vector.

- **std::vector<Teamcenter::DeepCopyData*>**

  Deep copy data vector.

- **tag_t**

  Tag to an object.

- **Teamcenter::DateTime**

  Class that represents the date in the Teamcenter server.

- **Teamcenter::OperationDispatcher**

  Helper methods for extension framework.

- **Teamcenter::Soa::Common::ObjectPropertyPolicy**

  Service-oriented architecture (SOA) representation of property policies.

- **Teamcenter::Soa::Server::InvalidCredentialException**

  Service-oriented architecture (SOA) exception error for incorrect ID.

- **Teamcenter::Soa::Server::ModelSchema**

  Service-oriented architecture (SOA) representation of schema.

- **Teamcenter::Soa::Server::PartialErrors**

  Service-oriented architecture (SOA) class for holding partial errors.

- **Teamcenter::Soa::Server::Preferences**

  Service-oriented architecture (SOA) class for holding preferences.

- **Teamcenter::Soa::Server::ServiceData**

  Service-oriented architecture (SOA) class that holds model objects and partial errors.

- **Teamcenter::Soa::Server::ServiceException**

  Service-oriented architecture (SOA) service exception class.

- **Teamcenter::Soa::Server::TypeSchema**

  Service-oriented architecture (SOA) representation of schema information.

Primitive data types reference

Primitive data types are generic data types such as character, double, float, Boolean, and short.

When you **create a business object operation** or **service operation**, you can choose a primitive data type as the parameter type.

To access the primitive data types, open the **Extensions\Code Generation\Data Types\Primitive Data Type** folders. Following are the standard primitive data types:

- **bool**

  Allows two choices to the user (true or false).

- **char**

  A single character, such as **A**, **B**, **Z**.

- **double**

  A double-precision floating point decimal number. (For Oracle, the limit for the double property value is 1E-130 to 9E125. For SQL Server, the limit is 2.3E-308 to 1.7E308.)

- **float**

  A 4-byte decimal number from the range 3.4E to 38.

- **int**

  An integer without decimals from 1 to 999999999.

- **long**

  A string of unlimited length.

- **void**

  Associated with no data type.

Template data type reference

Template data types allow code to be written without consideration of the data type with which it is eventually used. You can use the template data types as the parameter type when you **create business object operations** and **service operations**.

To access the template data types, open the **Extensions\Code Generation\Data Types\Template Data Type** folders. Following are the standard template data types:

- **std::map**

  Map from the standard namespace.

- **std::set**

  Set from the standard namespace.

- **std::vector**

  Vector from the standard namespace.

Operations

Add a business object operation

Operations are actions you can perform on business objects and properties in Teamcenter. You can create operations on either COTS or custom business objects and properties. To see these operations, right-click a business object, choose **Open**, and in the resulting editor, click the **Operations** tab.
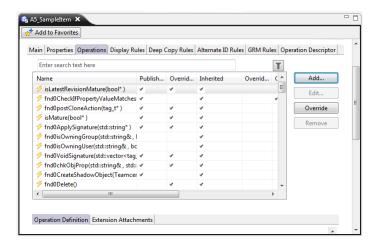
Operations have an operation template. The template defines the basic signature API structure of the operation: parameters, inputs, and outputs. An operation template can therefore be used to define new operations that follow the same signature pattern without having to redefine the signature again.

A business object operation is a function on a business object. It is defined with parameters, a return value, inheritance, and whether it is overridable. After you create a business object operation, you must generate code and write an implementation for the operation. You can only write operations on children of the **POM_application_object** business object.

You can also **create a property operation** or **operations on services**.

1. Ensure that you have set the proper active release for the operation. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.
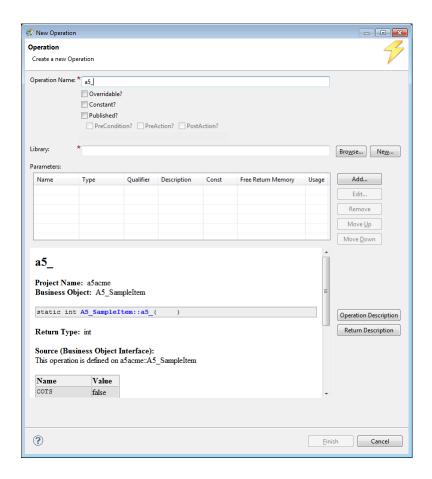
2. Set the active library for the operation. Open the **Extensions\Code Generation\Libraries** folders, right-click the custom library, and choose **Organize**→**Set as active Library**. A green arrow in the library symbol indicates it is the active library.

3. In the **Business Objects** folder, browse to the business object to which you want to add the operation. To search for a business object, click the **Find** button 🔍 at the top of the view.

4. Right-click the business object, choose **Open**, and click the **Operations** tab in the resulting view.

5. Click the **Operations** tab and click the **Add** button.

> **Note:**
>
> The **Add** button is available only after you **create a release**.

The Business Modeler IDE runs the New Operation wizard.

6. Perform the following steps in the **Operation** dialog box:

a. In the **Operation Name** box, type a name for the operation.

When you name a new data model object, a prefix from the template is automatically affixed to the name to designate the object as belonging to your organization, for example, **a4_**. The prefix for operations starts with a lowercase letter.

b. Select the following check boxes that apply:

• **Overridable?**

Allows child business objects to override this method.

• **Constant?**

Sets the return constant value to **const**.

• **Published?**

Declares the operation as published for use.

- **PreCondition?**

  Places limits before an action.

- **PreAction?**

  Executes code before an action.

- **PostAction?**
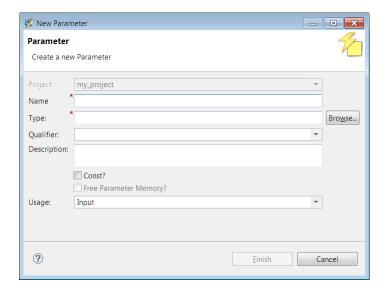
  Executes code after an action.

c. Click the **Browse** button to the right of the **Library** box to select the library to hold the source for the operation.

d. To set parameters on the operation, click the **Add** button to the right of the **Parameters** table. These parameters are the arguments to the C++ functions.

   The New Parameter wizard runs.

e. Perform the following steps in the **Parameter** dialog box:

   A. In the **Name** box, type a name for the parameter.

   B. Click the **Browse** button to the right of the **Type** box to select a data type to hold the parameter.

      The Find Data Type wizard runs.

C. In the **Choose a Data Type Object** dialog box, you can select the following data types to filter:

- **Primitive**

  Generic data types. Only **boolean**, **double**, **float**, and **int** are available for services.

- **External**

  Standard data types. Only those listed are available for services, and they are defined by Teamcenter.

- **Template**

  Data types that allow code to be written without consideration of the data type with which it is eventually used. Only **vector** is available for services.

- **Interface**

  Data types for returning business objects. Only business objects with their own implementation are selectable. For business objects without their own implementation, the closest parent can be selected instead.

- **Generated**

  Service data types, such as structure, map, and enumeration data types. Only the data types set up for this service are shown as being available for this service. You can manually type the name of a type that is in another service as long as it is in the same service library.

D. Click **Finish** in the **Choose a Data Type Object** dialog box.

E. Click the arrow in the **Qualifier** box to select the qualifier for the return value.

- No value

  Indicates that no parameter is passed.

- **&**

  Indicates that the parameter is passed by reference. The actual parameter is passed and not just the value of the parameter.

- *****

Indicates that the address of the parameter is passed. The address is the pointer to the parameter.

- ******

  Indicates the address of the pointer to the parameter is passed. The address is a pointer to the pointer.

- **[ ]**

  Indicates that the parameter is an array. The address of the first element in the array is passed.

- ***&**

  Indicates a parameter instance is created within the routine and the pointer of the instance is returned. The caller must manage the return pointer.

F. In the **Description** box, type a description for the parameter.

G. Select the **Const?** check box if the parameter is a constant.

H. Select the **Free Parameter Memory?** check box if the caller has to free the memory. This applies only to output and input/output parameters.

I. Click the arrow in the **Usage** box select whether the parameter is an input, output, or input/output parameter.

J. Click **Finish**.

   The **Parameters** table displays the new parameter.

f. In the **Operation Description** box, type a description of the new operation.

g. In the **Return Description** box, type a description of the return value for the operation.

h. Click **Finish**.

   The Business Modeler IDE displays the new operation in the **Operations** editor.

7. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 🖫 on the main toolbar.

8. Generate code.

   Right-click the **Business Objects** folder and choose **Generate Code→C++ Classes**.

9.   **Write an implementation** in the generated *business_object***Impl.cxx** file.
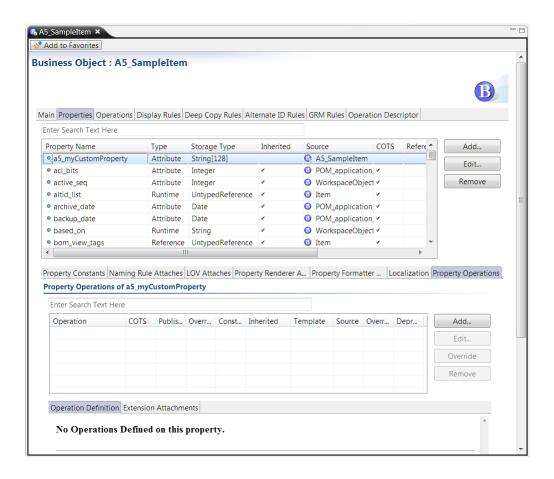
Add a property operation

Operations are actions you can perform on business objects and properties in Teamcenter. You can create operations on either COTS or custom business objects and properties. To see business object operations, right-click a business object, choose **Open**, and in the resulting editor, click the **Operations** tab; to see property operations, click the **Properties** tab, select a property, and click the **Property Operations** tab.

A property operation is a function on a property. You can publish getter and setter methods on a custom operation. After you create an operation on a property, you must generate code and write an implementation for the operation. You can also add a property operation when you create a persistent property.

You can also create a **business object operation** or **operations on services**.

1.   Ensure that you have set the proper active release for the operation. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2.   Set the active library for the operation. Open the **Extensions\Code Generation\Libraries** folders, right-click the custom library, and choose **Organize→Set as active Library**. A green arrow in the library symbol indicates it is the active library.

3.   Select the business object whose property you want to add an operation to. In the **Business Objects** folder, browse to the business object. To search for a business object, click the **Find** button 🔍 at the top of the view.

4.   Right-click the business object, choose **Open**, and click the **Properties** tab in the resulting view.

5.   Select a property in the properties table and click the **Property Operations** tab.
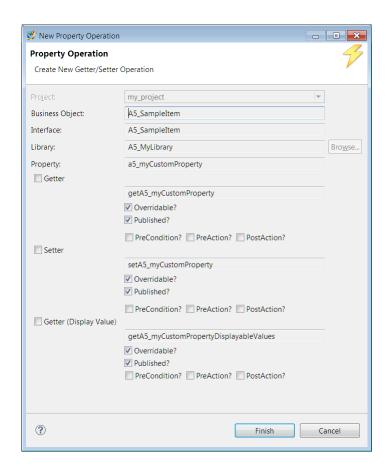
6. Click the **Add** button on the **Property Operations** tab.

> Note:
>
> The **Add** button is enabled only after you have **created a release**.

The Business Modeler IDE runs the **New Property Operation** wizard.

7. In the **Property Operation** dialog box, select the following check boxes that apply:

   • **Getter**

   Enables getting the value of the property.

   • **Setter**

   Enables setting the value of the property.

   • **Getter (Display Value)**

   Enables getting the display name of the property.

   This getter uses the **getDisplayableValues** property operation. The name of the function has the following format:

   get*property-name***DisplayableValues**

8. Select the following check boxes that apply:

   • **Overridable?**

Allows child business objects to override this method.

- **Published?**

  Generates the getter or setter method.

- **PreCondition?**

  Places limits before an action.

- **PreAction?**

  Executes code before an action.

- **PostAction?**

  Executes code after an action.

  When done making changes, click **Finish**.

  The new getter/setter methods display on the property. Click the method to see the operation definition.

9. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

10. Generate code.

    Right-click the **Business Objects** folder and choose **Generate Code→C++ Classes**.

11. **Write an implementation** in the generated *business_object* **Impl.cxx** file.

Override an operation

You can make a business object or property operation overridable by other operations on child business objects. An operation is overridable if it is inherited from a parent business object, marked as overridable, is not already overridden. The **Override** button is available if these conditions are met.
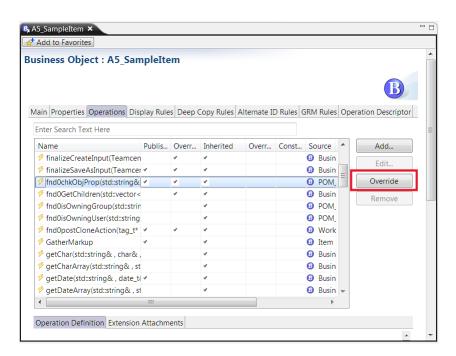
To override a business object operation, in the **Operations** tab, select the operation and click the **Override** button. To override a property operation, in the **Property Operations** tab, select the operation and click the **Override** button. The button changes to **Remove Override**.

1. Select the business object or property whose operation you want to override.

2. For a business object, click the **Operations** tab. For a property, click the **Property Operations** tab.
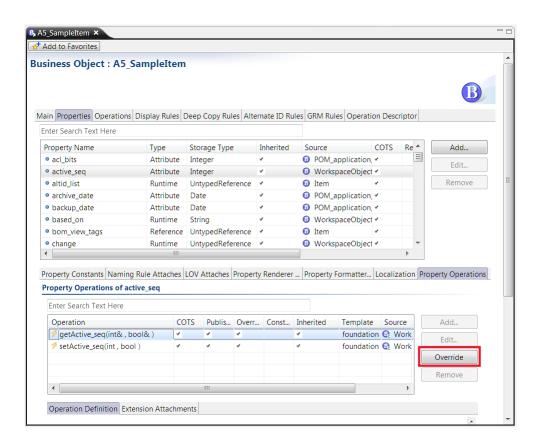
3. Select the operation you want to override in the **Operations** or the **Property Operations** tab.

   The **Override** button is available if the operation is inherited from a parent business object, is marked as overridable, and is not already overridden. The **Remove Override** button displays if the operation is already overridden.

   The following example shows selecting a business object operation to override.
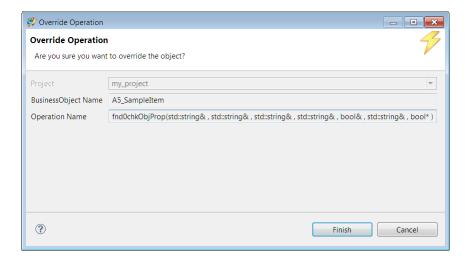
The following example shows selecting a property operation to override.

4. To override the operation, click the **Override** button.
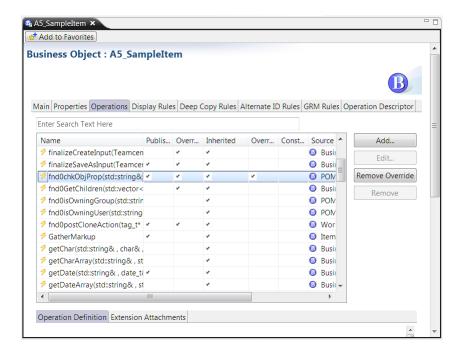
   The Override Operation wizard runs.
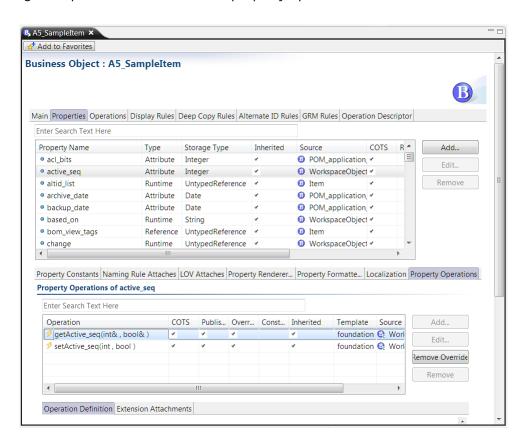
5. Click **Finish**.

   The button changes to **Remove Override**. To remove an override, click the **Remove Override** button.

The following example shows an overridden business object operation.

The following example shows an overridden property operation.

6. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

7. Right-click the **Business Objects** folder and choose **Generate Code→C++ Classes**.

8. In the generated **Impl** file, write the implementation code. Point to the new operation to use instead of the overridden operation.

Deprecate an operation

You can deprecate custom business object operations and custom property operations. *Deprecation* means that an object will be deleted in a future release.

To deprecate a business object operation, in the **Operations** tab, select the operation and click the **Deprecate** button. To deprecate a property operation, in the **Property Operations** tab, select the operation and click the **Deprecate** button. The button changes to **Undeprecate**.

1. Ensure that the deprecation policy is set.

   Your company has its own deprecation policy that states the number of releases that must pass after an object's deprecation before the object can be deleted. To change the number of releases, as well as whether a deprecation policy is set for the project, you must change the project properties.
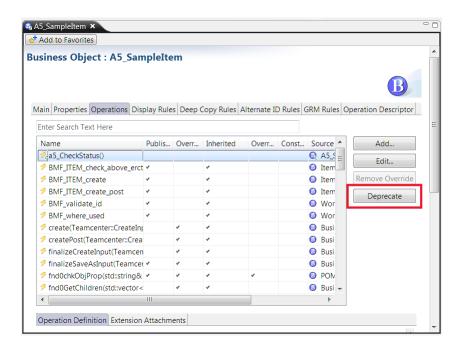
   a. Right-click the project, choose **Properties**, and choose **Teamcenter→Code Generation** in the left pane.

   b. Select the **Enable Deprecation Policy** check box to allow for removal of obsolete objects from the project.

   c. Click the arrow in the **Number of Allowed Releases before Deletion** box to select how many releases before objects can be deleted from the project.

   d. Click **OK**.

2. Select the business object or property whose operation you want to deprecate.

3. For a business object, click the **Operations** tab. For a property, click the **Property Operations** tab.

4. Select the operation you want to deprecate in the **Operations** or the **Property Operations** tab.

   The **Deprecate** button is displayed.
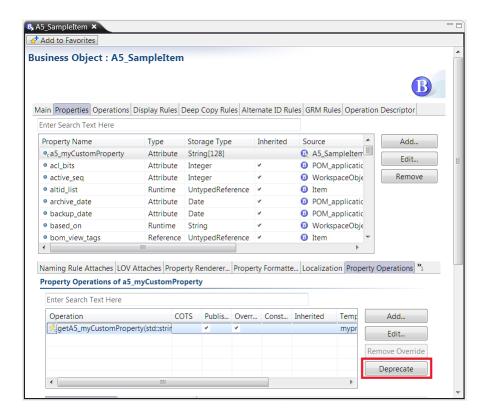
> Note:
>
> The **Deprecate** button is available only if the deprecation policy is set for the project, the operation is custom, and the operation was created in an earlier release. If not, a **Remove** button is displayed instead. An **Undeprecate** button displays if the operation has already been deprecated.

Following is an example of selecting a business object operation to deprecate.
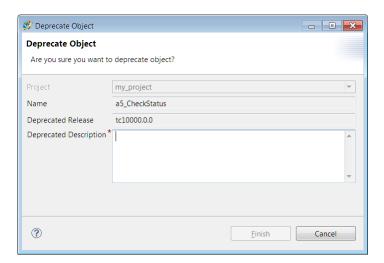
Following is an example of selecting a property operation to deprecate.

5.   Click the **Deprecate** button.
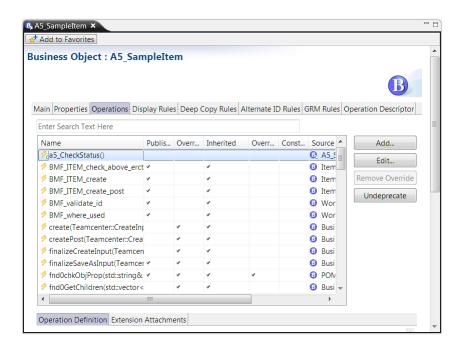
     The **Deprecate Object** dialog box is displayed.
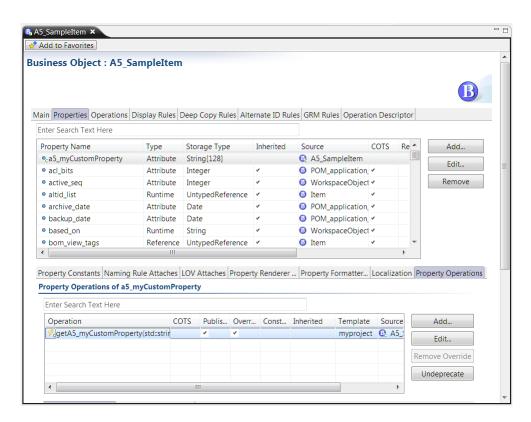
6.   In the **Deprecate Object** dialog box, type a description in the **Deprecated Description** box and click **Finish**.

     When an operation is deprecated, the button changes to **Undeprecate**.

     Following is an example of a deprecated business object operation.

Following is an example of a deprecated property operation.

Create an extension that uses a custom operation

An extension can be assigned to a business object or property so that it is invoked at a particular point (on a precondition, preaction, or postaction on the object). For business objects, this is done through

the **Extensions Attachment** tab in the **Operations** tab. For properties, this is done through the **Extensions Attachment** tab in the **Property Operations** tab.

1. **Create a custom operation**.

    When you create a new operation, ensure that you select the **Published** check box and select the action where you want to attach the operation (**PreCondition**, **PreAction**, and/or **PostAction**).
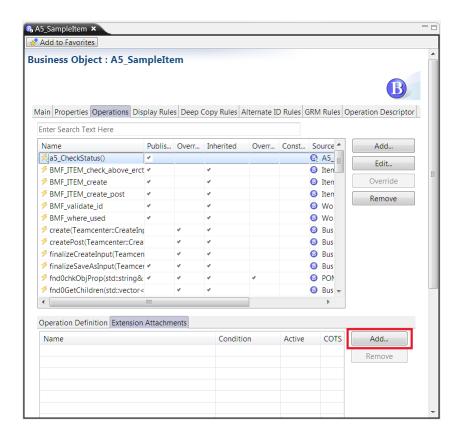
    Also ensure that you enter the parameters to the C++ functions in your implementation.

2. Create the extension that calls the operation.

    a. In the **Extensions** folder, right-click the **Rules→Extensions** folder and choose **New Extension Definition**.

    b. In the **Extension** dialog box, click the **Add** button to the right of the **Availability** table.

    c. In the **New Extension Availability** dialog box, click the **Browse** button to the right of the **Business Object Name** box and select the business object with the custom operation you want to use.

    d. In the **Operation Name** box, select the custom operation.

    e. In the **Extension Point** box, select the extension point that was set up when the custom operation was created (**PreCondition**, **PreAction**, and/or **PostAction**).

3. Add the extensions rule.

    a. To add the extension to a business object, open the business object and select the custom operation on the **Operations** tab. To add the extension to a property, select the property and select the operation on the **Property Operations** tab.

    b. Click the **Extension Attachments** tab.

    c. Choose the point at which you have made the operation available (**Pre-Condition**, **Pre-Action**, **Base-Action**, or **Post-Action**).
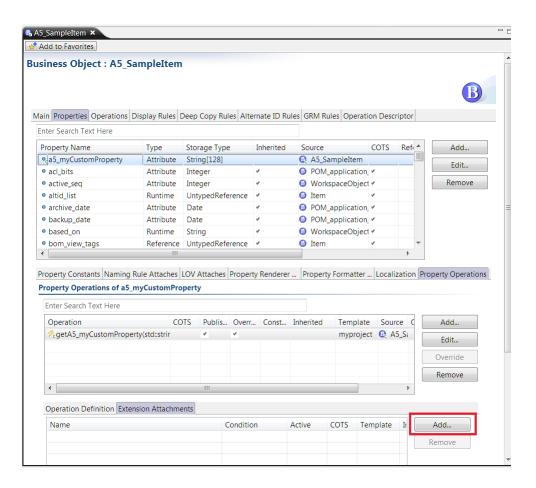
        The **Add** button is enabled only on the available points.

    d. Click the **Add** button to add the extension rule.

        Following is an example of adding an extension to a business object operation.

Following is an example of adding an extension to a property operation.

Now the operation is run when the extension is invoked at a precondition, pre action, or postaction.

For example, you can write an operation (such as the **hasValidSAPPartNumber operation example**) that checks to make sure that items have valid SAP numbers, and invoke the extension as a precondition for checkin.

Define a getter operation for a custom run-time property

Currently, the Business Modeler IDE does not allow you to define the getter operation for a custom property defined on a COTS business object. To overcome this limitation, you can manually add the following elements into your template to define the getter operation for your custom property.

In this example, the custom property name is **MyCustomObject**. This property is a single-value reference property. Therefore, the **PROP_ask_value_tag** operation is used.

- Add the following **TcOperationAttach** code to attach the **PROP_ask_value_tag** getter operation to the **MyCustomObject** property on the **UserSession** business object:

```
<TcOperationAttach operationName="PROP_ask_value_tag"
extendableElementName="UserSession"
        extendableElementType="Type" propertyName="MyCustomObject" description="">
      <TcExtensionPoint extensionPointType="PreCondition" isOverridable="true"/>
```

```
            <TcExtensionPoint extensionPointType="PreAction" isOverridable="true"/>
            <TcExtensionPoint extensionPointType="BaseAction" isOverridable="false"/>
            <TcExtensionPoint extensionPointType="PostAction" isOverridable="true"/>
        </TcOperationAttach>
```

- Add the following **TcExtension** code to define an extern function for the **MyCustomObject** property. The name of the extern function is **getMyCustomObjectBase**, which must be unique.

```
    <TcExtension name="getMyCustomObjectBase" internal="false"
          cannedExtension="false" languageType="CPlusPlus"
    libraryName="libMyCustomLibrary"
    description="">
          <TcExtensionValidity
    parameter="PROPERTY:UserSession:MyCustomObject:PROP_ask_value_tag:4"/>
      </TcExtension>
```

Change **libMyCustomLibrary** to the name of your library.

- Add the following **TcExtensionAttach** code to attach the external function as a **BaseAction** type of the getter operation:

```
    <TcExtensionAttach extensionName="getMyCustomObjectBase"
          operationName="PROP_ask_value_tag" isActive="true" propertyName="MyCustomObject"
          extendableElementName="UserSession" extendableElementType="Type"
    extensionPointType="BaseAction"
          conditionName="isTrue" description=""/>
```

Implement a getter base function for custom run-time property

The getter base function for a property operation is declared in the following manner to avoid C++ name mangling. In this example, the library name is **MyLib**:

```
    #ifdef cplusplus
    extern "C"
    {
    #endif
    extern MYLIB_API int getMyCustomObjectBase(METHOD_message_t *, va_list
    args);
    #ifdef cplusplus
    }
    #endif
```

The following is sample code for the **getMyCustomObjectBase** base function, which is expected to return the tag of the singleton instance of the custom business object. The instance is created but not saved so that this custom business object is actually used as a run-time object providing the operation codes, causing no impact on database data. The tag of the instance is cached in a static variable to make sure only one instance is created.

```
    int   getMyCustomObjectBase(  METHOD_message_t *, va_list args )
    {
```

```
va_list largs;
va_copy( largs, args );
va_arg( largs, tag_t ); /*Property Object tag_t  not used*/
tag_t*  customObjTag     = va_arg( largs, tag_t* );
va_end( largs );

int ifail = ITK_ok;
*customObjTag = NULLTAG;

//Create and cache an instance of my custom business object
static tag_t myCustomObjectTag = NULLTAG;
if( myCustomObjectTag == NULLTAG )
{
static const char *  customBOName  = "MyCustomBOName";
Teamcenter::CreateInput* creInput =
dynamic_cast<Teamcenter::CreateInput*>(Teamcenter::BusinessObjectRegistry
::
instance().createInputObject(customBOName,
OPERATIONINPUT_CREATE));

Teamcenter::BusinessObject*  obj  =
Teamcenter::BusinessObjectRegistry::instance().createBusinessObject(creIn
put);

myCustomObjectTag = obj->getTag();
}
*customObjTag = myCustomObjectTag;
return ifail;
}
```

Boilerplate code

Introduction to generating boilerplate code
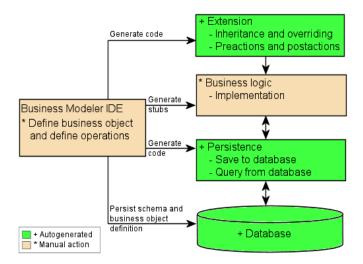
Generating boilerplate code can save time and effort over manually setting up the source files, and it can reduce coding errors. The autogenerated code conforms to a coding standard, therefore maintaining consistency. You can create C++ boilerplate source code files into which you can add implementation code by right-clicking the **Business Objects** folder and choosing **Generate Code→C++ Classes**.

> Note:
>
> **Generating code for services** is a different process. Right-click the **Code Generation→Services** folder and choose **Generate Code→Service Artifacts**.

Most of the code pieces are autogenerated; you only need to write code in the implementation (**Impl**) file, as shown in the following figure. First you create the business objects and operations, and their definitions are deployed to the database without writing any code. Then you generate code, creating

boilerplate files that provide inheritance and overriding behavior to the business object. Finally, you write the custom logic for the operation in the stub provided in the **Impl** file.

Generate C++ code

After operations are defined, you can create C++ boilerplate source code files into which you can add implementation code by right-clicking the **Business Objects** folder and choosing **Generate Code→C++ Classes**. This generates the C++ interface for the business object and other boilerplate code. A stub for the implementation class also is generated the first time (**Impl** file). Add the business logic manually in this class.

If you have any server code customizations from previous versions, you must regenerate the code and rebuild your libraries.

> Tip:
>
> If additional operations are added to the business object at a later time, the operations must be added manually to this implementation class.
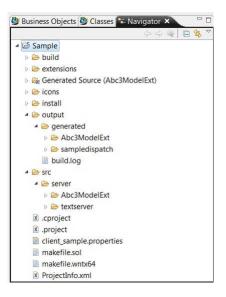
1. Create a **business object operation**, **property operation**, a **service operation**, or **extension**.

2. Ensure that the code generation environment is set up correctly. Right-click the project, choose **Properties**, and choose **Teamcenter→Build Configuration** and **Teamcenter→Code Generation**. Ensure that the project has the correct Teamcenter installation and compiler home set.

3. To generate boilerplate code for business object and properties operations, right-click the **Business Objects** folder and choose **Generate Code→C++ Classes**.

    > Caution:
    >
    > You *must* right-click the **Business Objects** folder and choose **Generate Code→C++ Classes** to generate code.

In addition to generating source code files for the customization, the generate process also creates makefiles to compile the source code. There is a platform-specific makefile created in the project's root folder (**makefile.wntx64**) and there are also a number of platform neutral makefiles created under the build folder for each library that is defined in the template. All of these makefiles may be persisted and are only updated by the Business Modeler IDE when something changes (a new library defined, build options changed, and so on)

All generated files are placed in the **output/generated** folder, with a subfolder for each library. The implementation stub files are placed under the **src/server** folder, with a subfolder for each library.

Note:

The code generated from the Business Modeler IDE ignores the **Published** flag set on the operation. In Teamcenter, every operation is a published operation. The published/ unpublished flag is provided so that users can tag their operations as published or unpublished.

There is no specific workaround for this problem as it does not cause any error in Business Modeler IDE or the code generated.

To generate boilerplate code for service operations, right-click the service containing the service operation and choose **Generate Code→Service Artifacts**. Check the **Console** view to see the progress of the generation.

4. After you generate code, open the **Advanced** perspective by choosing **Window→Open Perspective→Other**.

    The project folder is now marked with a **C** symbol. This indicates that the project has been converted to a CDT nature project. The C/C++ Development Toolkit (CDT) is a collection of Eclipse-based features that provides the capability to work with projects that use C or C++ as a

programming language. To learn more about CDT, in the top menu bar, choose **Help→Help Contents** and choose the *C/C++ Development User Guide* in the left pane of the **Help** dialog box.

Choose the **Project** menu in the top menu bar to see that build options are now added to the menu. Right-click the project in the **Navigator** view to see options added to the bottom of the menu for building and running the project in the context of a CDT.
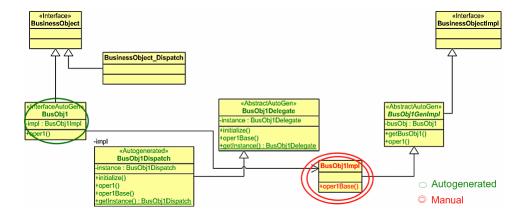
5. For business object and property operations, **write your operation implementation** in the generated *business_object***Impl.cxx** file. For services, **write the implementation** in the generated *service***Impl.cxx** files.

The Teamcenter *C++ API Reference*, available on Support Center, documents APIs in C++ signatures.

Generated business object classes

When an operation is added to a business object and the code generation command is executed, the following classes are generated:

- **BusinessObject** interface that defines the API for the business object. All callers work only with the interface.

- **Dispatch** class that implements the interface. It provides the overriding and extensibility capabilities.

- An abstract generated implementation class. This class provides getters and setters for the attributes on the business object and any other helper methods needed by the implementation.

- **Impl** class that is the actual implementation where the business logic is manually implemented by the developer.

- Delegate class that delegates invocations to the implementation class and breaks the direct dependency between the dispatch and implementation classes.

Class	Autogenerated	Purpose
Interface	Yes	C++ interface class. Defines published/ unpublished API. All server caller code dependent on this business object works with only this interface.
Dispatch	Yes	Implements the interface. Provides the overriding and extensibility (pre/post) capabilities to the business object.
Generated Implementation (**GenImpl**)	Yes	C++ abstract class. This class provides methods to get/set attribute values in the database and any other helper methods needed by the implementation.
Implementation (**Impl**)	No	Implementation class. The developer implements the core business logic for the business object in this class.
Delegate	Yes	C++ abstract class. This class breaks the direct dependency between the dispatch and implementation classes.

Implementation code

Write implementation code for a business object or property operation

You can write implementation code for a business object or property operation by using boilerplate source code files. Write an implementation after you have **created an operation** and have generated the boilerplate code.

The *C++ API Reference* available on Support Center documents APIs in C++ signatures.

1. Create the operation for which you need to write the implementation.

    *   **Business object operation**

        Right-click the business object against which you want to write the operation, choose **Open**, click the **Operations** tab in the resulting view, and click the **Add** button.

    *   **Property operation**

Right-click the business object against which you want to write the property operation, choose **Open**, select the property against which you want to write the operation, click the **Property Operations** tab, and click the **Add** button.

2. **Generate the boilerplate code**.

   Right-click the **Business Objects** folder and choose **Generate Code**→**C++ Classes**.

3. Locate the generated code files.

   By default, the generated files are saved in the **Project Files\src\server**_library_ folder. The _library_ folder holds _business_object_**Impl.cxx** files where you write business logic. (The **Generated Source Folder** folder holds files containing boilerplate code. You should never edit these files.)

   > Note:
   >
   > To change the location where generated code is saved, open the project's properties. Right-click the project, choose **Properties**, and select **Teamcenter**→**Code Generation**.

4. Switch to the **C/C++** perspective by choosing **Window**→**Open Perspective**→**Other**→**C/C++**.

5. Open the code file.

   Right-click the **src\server**_library_****_business_object_**Impl.cxx** file and choose **Open** or **Open With**.

   The file opens in an editor.

6. Write the implementation.

   Add your **implementation** at the bottom of the _business_object_**Impl.cxx** file where it contains the following text:

   ```
   ** Your Implementation **
   ```

7. **Build the libraries containing server code**.

8. Package your changes into a template.

   > Note:
   >
   > The Business Modeler IDE packages the built C++ library as part of its template packaging. The complete solution that includes the data model and the C++ run-time libraries are packaged together.

9. Install the package to a server.

Sample implementation code

When you create custom operations, you must **write implementation code** in the generated **Impl** class file.

The *C++ API Reference* documents APIs in C++ signatures.

> Note:
>
> To access the *C++ API Reference*, go to Support Center.

Following is a sample customization:

1. Create a custom child of the **Item** business object named **CommercialItem**.

2. Add a custom string property called **SAPPartNumber**.

3. On the **Operation Descriptor** tab, configure the **CreateInput** attributes to make the new property visible in the user interface during object creation (when users choose **File→New** in the rich client).

4. **Add a property operation** to the **SAPPartNumber** property to set the getter and setter code generation for the property.

5. **Add a new operation** to the **CommercialItem** business object called **hasValidSAPPartNumber**.

Given this example, following is some sample implementation code that could be written for it. This sample code is for demonstration purposes only.

- The following code block demonstrates how interface methods and ITKs can be called when adding a new operation on the business object.

```
/**
 * checks if the BusinessObject has a valid SAPPartNumber
 * @param valid – true if the SAPPartNumber is valid.
 * @return – returns 0 is executed successfully.
 */
int  CommercialItemImpl::hasValidSAPPartNumberBase( bool &valid )
{
    //get the handle to the interface object first.
    CommercialItem *cItem = getCommercialItem();

    //invoke method on the Interface ( CommericalItem )
    ifail = cItem->getSAPPartNumber(sapPartNumber, isNull);

    //Write code here to validate the sapPartNumber value
    //agaist Business Requirement and update "valid" variable.

    //You can also invoke ITK using the tag.
    //Use getTag() to get the tag to be passed on to the ITK.
```

```
    if( ifail == ITK_ok )
    {
        tag_t cItemTag = cItem->getTag();
        //You may write custom code involving ITK calls with input as cItemTag here.
    }

    return ifail;
//-------------------------------------------------------------------------------
// CommercialItemGenImpl::getSAPPartNumberBase
// Base for Accessor of SAPPartNumber
//-------------------------------------------------------------------------------
int CommercialItemImpl::getSAPPartNumberBase( std::string &value, bool &isNull ) const
{
    int ifail = ITK_ok;

    //Add custom code here if needed.

    ifail = CommercialItemGenImpl::getSAPPartNumberBase( value, isNull );

    //Add custom code here if needed.
    return ifail;
}
```

- The following code block demonstrates how a super method is called when overriding an existing operation from a child business object, and how a getter operation is added to a property. Normally the super method should be invoked at the beginning of the overridden method. However, this does not apply to post methods. The following sample method is a postaction of create and therefore requires that the **super_createPostBase** method be called at the end of the sample.

```
/**
 * desc for createPost
 * @param creInput - Description for the Create Input
 * @return - return desc for createPost
 */
int  CommercialItemImpl::createPostBase( Teamcenter::CreateInput *creInput )
{
    int ifail = ITK_ok;

     // In addition to the implementation in parent BusinessObject write
     // your specific Implementation
     if(ifail = ITK_ok)
     {
        //Add custom code here if needed.
        /*
        //use getters methods available on OperationInput/CreateInput to access the
      //values in CreateInput.
      //for example below code gets the SAPPartNumber set by the user during
        //the create CommercialItem Action on the create UI.

        std::string sAPPartNumber ;
        bool isNull = false ;
        ifail = creInput->getString("SAPPartNumber",sAPPartNumber,isNull);

        //Implement custom code to meet the Business Requirement
        */
     }

    //Add custom code here if needed.
```

```
 /*
           Custom code
*/
      // Call the super createPost to invoke parent implementation
  ifail = CommercialItemImpl::super_createPostBase(pCreateInput);

  return ifail;

}
```
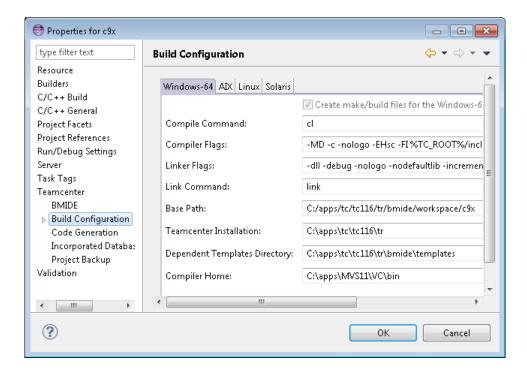
Server code

Build server code on Windows

1. Ensure that the build configuration is set properly.

    a. Right-click the project and choose **Properties**.

    b. Choose **Teamcenter→Build Configuration** in the left pane of the **Properties** dialog box.

    The **Build Configuration** dialog box is displayed.

    A. Verify that the build flags in the lower pane are set properly for the platform you are building on. By default the libraries are built in release mode.

    You can change these flags to build in debug mode. Use the **TC_ROOT***sample* *compile.bat* script for the required flags.

       B.   Click the **Browse** button to the right of the **Compiler Home** box to select the location where your C++ compiler is located. For example, if your platform is Windows and you are using Microsoft Visual Studio, browse to *compiler-install-path***\VC\bin**.

       c.   Click **OK**.

2.   Generate code by right-clicking the **Business Objects** folder and choosing **Generate Code→C++ Classes**.

After code is generated, a **C** symbol is placed on the project folder. This indicates that the project has been converted to a CDT (C/C++ Development Tools) project. The C/C++ Development Toolkit (CDT) is a collection of Eclipse-based features that provides the capability to work with projects that use C or C++ as a programming language. You can work on the project in the CDT perspective by choosing **Window→Open Perspective→Other→C/C++**.

After a project is converted to CDT, the **Project** menu in the top menu bar displays the following build options.

Menu command	Description
Build All	Builds all projects in the workspace. This is a full build; all files are built.
Build Configurations	Allows you to manage build configurations.
Build Project	Builds the currently selected project. This is a full build; all files in the project are built.
Build Working Set	Builds the current working set.
Clean	Runs the make clean command defined in the makefile. Try this option and then generate C++ classes again if you're experiencing a problem building.
Build Automatically	When checked, the CDT performs a build whenever a file in a project is saved. You should turn off this feature for very large projects.
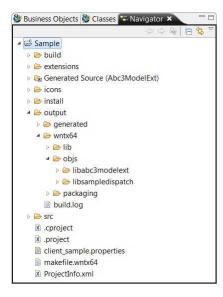
To learn more about these menu options, choose **Help→Help Contents** in the top menu bar and choose the *C/C++ Development User Guide* in the left pane of the **Help** dialog box.

3.   **Write your code in the implementation files**.

4. While in the **C/C++** perspective, choose **Project→Build Project** to build the libraries containing server code.

    If **Project→Build Automatically** is already selected, code is built automatically any time the code is changed and saved.

    The compiled artifacts are placed in the **output/wntx64** folder. Each defined library has a subfolder under the **output/wntx64/objs** folder for the intermediate object files (.obj) and all compiled libraries (.dll and .lib) are placed in the **output/wntx64/lib** folder.

> Caution:
>
> You *must* build on a Windows client if the code is to be used on a Windows server or build on a Linux client for a Linux server. If the packaged library files are installed on the wrong platform, they do not work.
>
> Spaces in path names may cause issues in Linux or with a version of Windows that does not convert 8.3 file names. If you already have spaces and are experiencing issues, you may create symbolic links in Linux or use **junction** or **mklink -j** in Windows.
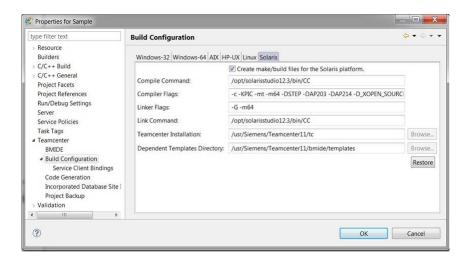
Build code on platforms not supported by the Business Modeler IDE

The build process in the Business Modeler IDE is designed to build server code on Windows platforms only. If you intend to build the same custom server code on multiple platforms, you can use the same project to generate the code and makefiles for each platform. You can use this process to build on the platforms that appear on the **Build Configuration** dialog box for your project.
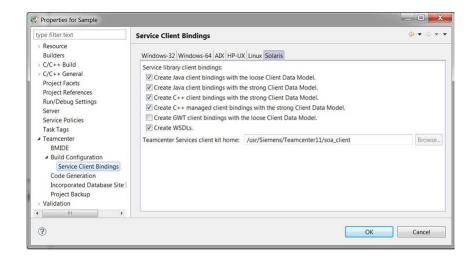
1. On Windows systems, create your new project in the Business Modeler IDE, and add new business objects, properties, and operations. Generate code and build server code.

2. Right-click the project and choose **Properties**. On the left side of the dialog box, choose **Teamcenter→Build Configuration**.

3. The dialog box includes a tab for each supported platform. For the platforms you want to build libraries on, select the **Create make/build files for the** *platform-name* **platform** ☑ check box.

4. Fill in the **Teamcenter Installation** and **Dependent Templates Directory** boxes for the appropriate locations relative to the target platform and host.

5. If you have defined services in the template, choose **Teamcenter→Build Configuration→Service Bindings**. Select the client bindings you want for each target platform, and fill in the **Teamcenter Services client kit home** and **Rich Client plug-in folder** boxes to the appropriate locations relative to the target platform and host.

   The next time you generate source code, either for metamodel customizations or for service operations, a platform specific makefile is also created (for example, **makefile.sol**).
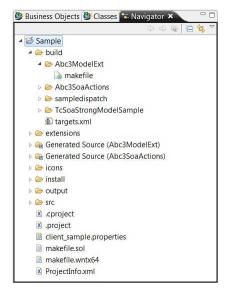
6. For the additional platforms, copy the new template directory and subdirectories, for example:

```
C:\apps\Teamcenter\bmide\workspace\release\MyTemplate
```

7. The makefile can be used on the target platform to generate code and build the libraries (for example, **make -f makefile.sol**). This generates all code for the project and compiles the libraries.

There is a platform-specific makefile created for each selected platform (for example, **makefile.wnti32**, **makefile.sol**, and so on). These files are created in the root folder of the project. There are also a number of platform-neutral makefiles created under the build folder for each library that is defined in the template. All of these makefiles may be persisted and are only updated by the Business Modeler IDE when something changes (a new library defined, build options changed, and the like). All platform-specific output files (such as .objs and .dll files) are created under the platform-specific output folder (for example, **output/wnti32**, **outputs/sol**, and so on).
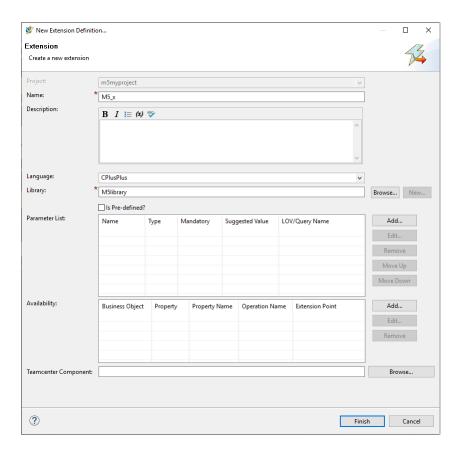
Extensions

Define an extension

An *extension* is a business rule that adds pre-defined behavior to a business object operation and fires as a pre-condition, pre-action, or post-action. When you define an extension, you identify the parameters passed to the extension when it is executed, and how the extension is made available to other business objects.

1. Expand the **Extensions>Rules** folders.

2. Right-click the 🗲 **Extensions** folder and choose **New Extension Definition**.

    The New Extension Definition wizard runs.

3. Specify options for the extension.

For this option	Do this
Name	Type a name for the extension. • If the programming language for the extension is C, the extension name must match the name of the function. • If the programming language for the extension is C++, the extension name must conform to the following format: *NameSpace*::*ClassName*::*MethodName*
Description	Type a description of the extension and how to implement it.
Language	From the list, choose the programming language of the function, either C or C++.
Library	Click **Browse** and select the library that contains the function. You can place libraries in the *TC_ROOT***bin** directory, or you can set the path to valid libraries with the **BMF_CUSTOM_IMPLEMENTOR_PATH** preference (*BMF* refers to the Business Modeler Framework). To access preferences, in the My Teamcenter application, choose **Edit→Options** and click **Search** at the bottom of the **Options** dialog box.

For this option	Do this
Is Pre-defined?	If you want to make the extension available to other templates that depend on this template, mark the checkbox.

Is Pre-defined?	Means this
Unchecked	The extension is visible and available for use ONLY within the current template. Any templates that depend on the current template will not see this extension and therefore cannot use it.
Checked	The extension is visible for the current template and all other templates that depend on the current template.
	Note that if the extension is marked as pre-defined, altering the extension could have undesirable ripple effects if the extension is used in other templates.

For this option	Do this
Parameter List	If there are parameters to be passed to the extension, click **Add**.

The **Extension Parameter** dialog box is displayed.

Specify options for the extension parameter:

For this option	Do this
Name	Type a name for the parameter.
Type	From the list, select the parameter type, one of **String**, **Integer**, or **Double**.
Mandatory	Select the check box if the parameter is mandatory (that is, the value for that parameter cannot be null).
Suggested Value	Select the type of value for the parameter: • **None** if you type the argument value manually when assigning the extension. • **TcLOV** if the values are derived from a list of values.

For this option	Do this

For this option	Do this
	If you select **TcLOV**, then for **LOV Name** click **Browse** and locate the list of values for the parameter. • **TcQuery** if the values are derived from the results of a saved query. If you select **TcQuery**, then for **Query Name** click **Browse** and locate the saved query. The Teamcenter Repository Connection wizard prompts you to log on to a server to look up its available saved queries.

	Click **Finish**. The parameter is added to the **Parameter List** table. To change the order in which the parameters are passed, select a parameter and click **Move Up** or **Move Down**.
Availability	To define availability of the extension to business objects, click **Add**. The **Extension Availability** dialog box is displayed. Specify options for the availability parameter:

For this option	Do this
Business Object Name	Click the **Browse** and choose the business object on which to place the extension. Availability defined for a business object applies to its children.

For this option	Do this		
	For this option	**Do this**	
		User exits also display in this list, so that you can make a user exit available for use by the new extension.	
	Business Object Or Property	• Select **Type** if the rule is to be associated with a business object. • Select **Property** if the rule is to be associated with a property on the business object. If you select **Property**, then for **Property Name** click **Browse** and select the property.	
	Operation Name	Click **Browse** and select the operation to place on the business object. The list includes both legacy and new operations for that business object. The new operations have the C++ style signature.	
	Extension Point	From the list, select the point as a **PreCondition**, **PreAction**, or **PostAction** action. Pre/post conditions for an operation are controlled during operation creation time. For example, if a precondition is enabled for the **XYZ(***data-types***)** operation during its creation time, then the availability wizard shows the precondition for that operation only.	
	Click **Finish**. The availability item is added to the **Availability** table.		
Teamcenter Component	(Optional) Click **Browse** and select a Teamcenter Component with which to associate this extension.		

4. Click **Finish**.

    The extension is added to the **Extensions** folder.

    Now that you have added the extension, you can assign it to business objects or business object properties.

5. To save the changes to the data model, choose **BMIDE**>**Save Data Model**, or on the main toolbar click **Save Data Model** 💾.
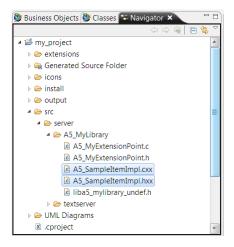
Write extension code

Extensions allow you to write a custom function or method for Teamcenter in C or C++ and attach the rules to predefined hook points in Teamcenter.

After you assign an extension to a business object or a property operation so that it is called at a particular point (on a pre-condition, pre-action, base action, or post-action on the object), you must write the code that takes place in Teamcenter when the extension is called.

1. Open the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2. In the **Extensions** view, open the **Rules\Extensions** folders, right-click the new extension you have created, and choose **Generate extension code**.

   The extension boilerplate code is generated into an *extension-name*.**cxx** C++ file and an *extension-name*.**hxx** header file. To see these files, open the project in the **Navigator** view and browse to the **src\server***library* directory.

> Note:
>
> You may need to right-click in the view and choose **Refresh** to see the files that were generated.

3. Write your extension code in the new *extension-name*.**cxx** and *extension-name*.**hxx** files. You can use the standard methods of writing code for Teamcenter.

4. **Build your libraries**.

5. Deploy the customization to a test server by choosing **BMIDE**→**Deploy Template** on the menu bar. Check to make sure the extension works properly.

Extension example: Add a postaction on a folder business object
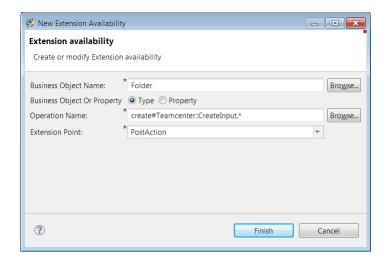
In this example, an extension named **H2FolderPostAction** is attached to the **Folder** business object and runs as a postaction when the user creates a folder. (**H2** in this example represents the project naming prefix.)

Although sample code for this extension is provided, you must write your own business logic in the extension that is specific to your business needs.

1.   Open the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2.   Define the **H2FolderPostAction** extension.

   a.   Create a library named **H2libext**.

   b.   Expand the project and the **Rules\Extensions** folders.

   c.   Right-click the **Extensions** folder and choose **New Extension Definition**.
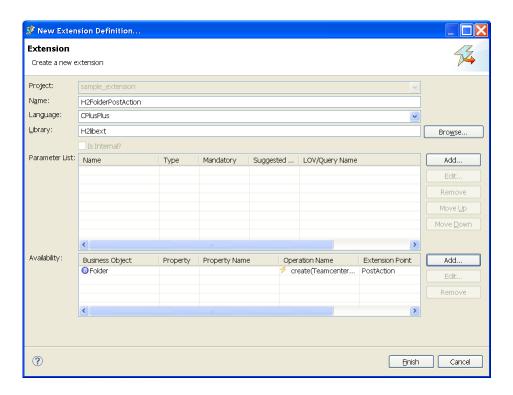
          The New Extension Definition wizard runs.

   d.   Perform the following In the **Extension** dialog box:

          A.   In the **Name** box, type **H2FolderPostAction**.

          B.   In the **Language** box, select **CPlusPlus**.

          C.   In the **Library** box, select **H2Libext**.

          D.   Click **Add** to the right of the **Availability** table and perform the following in the **Extension availability** dialog box:

                 i.   In the **Business Object Name** box, select **Folder**.

                 ii.   In the **Operation Name** box, select **create(Teamcenter::CreateInput*)**.

                 iii.   In the **Extension Point** box, select **PostAction**.
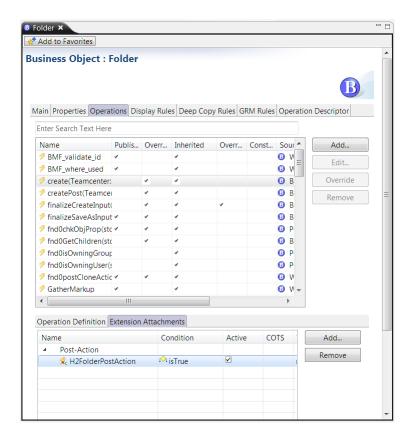
iv. Click **Finish** in the **Extension availability** dialog box.

The extension appears.

E. Click **Finish** in the **Extension** dialog box.

3. Attach the **H2FolderPostAction** extension to the **Folder** business object.

a. In the **Business Objects** view, right-click the business object you have made available on the extension, choose **Open**, and click the **Operations** tab in the resulting editor and click **create(Teamcenter::createInput*)**.

b. Click the **Extension Attachments** tab and click **Add**

c. In the extension attachment wizard, for the extension point select **PostAction** and for the extension select the **H2FolderPostAction** extension.

d. Click **Finish**.

The extension is added to the postaction.

The XML entries are created as follows:

```xml
<Add>

    <TcExtension name="H2FolderPostAction" internal="false" cannedExtension="false"
        languageType="CPlusPlus" libraryName="libH2libext" description="">
     <TcExtensionValidity parameter="TYPE:Folder:create#Teamcenter::CreateInput,*:3"/>
    </TcExtension>
    <TcExtensionAttach extensionName="H2FolderPostAction"
operationName="create#Teamcenter::CreateInput,*"
        isActive="true" extendableElementName="Folder" extendableElementType="Type"
        extensionPointType="PostAction" conditionName="isTrue" description=""/>

</Add>
```

The library file is saved as **libH2libext**. XML entries store the exact name of the library without the library extension (**.dll**, **.so**, and so on).

4. Implement the **H2FolderPostAction** extension.

   a. In the **Advanced Perspective,** right-click the **H2FolderPostAction** extension and choose **Generate extension code**.

      The extension boilerplate code is generated into a **H2FolderPostAction.cxx** C++ file and a **H2FolderPostAction.hxx** header file. To see these files, open the project in the **Navigator** view and browse to the **src\server\H2libext** directory.

> Note:
>
> You may need to right-click in the view and choose **Refresh** to see the files that were generated.

b. Open the **H2FolderPostAction.cxx** file in a C/C++ editor and add your custom business logic. Following is a sample file:

```
#include <H2libext/H2FolderPostAction.hxx>
#include <stdio.h>
#include <stdarg.h>
#include <ug_va_copy.h>
#include <CreateFunctionInvoker.hxx>
using namespace Teamcenter;
int H2FolderPostAction (METHOD_message_t* msg, va_list args)
{
        std::string  name;
        std::string description;
    bool isNull;
        // printf("\t +++++++++++  In H2FolderPostAction()  +++++++++++\n");

        va_list local_args;
        va_copy( local_args, args);

CreateInput *pFoldCreInput = va_arg(local_args, CreateInput*);

        pFoldCreInput->getString("object_name",name,isNull);

   pFoldCreInput->getString("object_desc",description,isNull);
   // Write custom business logic here

    va_end( local_args );

    return 0;
}
```

5. Build the library (**libH2libext.dll** file) for the **H2FolderPostAction** extension.

    a. Run the **dumpbin** script (Windows) or **nm** script (Linux) to see if the **H2FolderPostAction** extension symbol is exported correctly, for example:

        ```
        dumpbin /EXPORTS <path_of_libH2libextr>\libH2libext.dll
        ```

        The symbol should not be mangled (C++). The output should look similar to the following:

        ```
        11471 00002BE0 H2FolderPostAction = _H2FolderPostAction
        ```

    b. Put the **libH2libext.dll** library file in the *TC_ROOT***bin** directory, or from the rich client, set the preference **BMF_CUSTOM_IMPLEMENTOR_PATH** to the path of the library.

6. Deploy the **H2FolderPostAction** extension.

7. Test the **H2FolderPostAction** extension by creating a folder object from the rich client. The extension should get executed.

Workflow extension example

Workflow extension example: Overview

This custom extension example initiates a workflow process on an item revision when creating an item.

1. **Define the extension**.

2. **Assign the extension to an extension point**.

3. **Develop the C custom extension code**.

4. **Execute the extension**.

Workflow extension example: Define the sample extension in the Business Modeler IDE

1. Open the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2. Navigate to the **Extensions** view and open the **Rules**→**Extensions** folder. Right-click the **Extensions** folder and choose **New Extension Definition**.

   The New Extension Definition wizard runs.

3. Enter the name of the extension rule as *prefix***bmf_extension_workflow_sample**. The prefix corresponds to the file name prefix for your project, for example, **G4_**. This name must match the function name defined in the header file.

4. In the **Language Type** box, select **ANSI-C**.

5. Click the **Browse** button to the right of the **Library** box and select the **libsampleExtension** library.

6. Add the parameter definitions to this extension.

   a. Click the **Add** button in the **Parameter List** table.

      The **Extension Parameter** dialog box is displayed.

   b. Create the **Release Process** parameter.

      A. In the **Name** box, type **Release Process**.

      B. In the **Type** box, select **String**.

      C.   Select the **Mandatory** check box to make the parameter required.

      D.   Click **Finish**.

   c.   Click the **Add** button in the **Parameter List** table. Now create the **Company ID** parameter.

      A.   In the **Name** box, type **Company ID**.

      B.   In the **Type** box, select **Integer**.

      C.   Clear the **Mandatory** check box to make the parameter optional.

      D.   Click **Finish**.

7.   Modify the availability of this extension.

   a.   Click the **Add** button in the **Availability** table. The **Extension Availability** dialog box is displayed.

      A.   Click **Browse** in the **Business Object Name** box and select **Item**.

      B.   In the **Business Object or Property** area, select the **Type** check box.

      C.   In the **Operation Name** box, select **ITEM_create**.

      D.   In the **Extension Point** box, select **PostAction**.

      E.   Click **Finish**.

   b.   Click **Finish** in the **Extension** dialog box.

This extension is now available to the **ITEM_create** operation only as a post-action.

Workflow extension example: Assign the sample extension

1.   Open the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2.   In the **Business Object** view, right-click the **Item** business object, choose **Open**, and click the **Operations** tab.

A view displays the extension operations attached to the business object.

3.   Select the **ITEM_create** operation.

4.   Click the **Extension Attachments** tab and click the **Add** button.

The Add Extension Rule wizard runs.

5.   Select **PostAction** for the extension point.

Click the **Browse** button in the **Extension** box and select *prefix***bmf_extension_workflow_sample**. Add arguments in the **Arguments** box for **TCM Release Process** using a company ID of **56** and click **Finish**.

Note how the check box in the **Active** column shows how this extension is activated.

Workflow extension example: Develop the C custom extension code for the sample extension

This custom extension initiates a workflow process on an item revision (after it is created) when creating an item. This is custom code developed using standard ITK development practices.

1.   Open the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2.   In the **Extensions** view, open the **Rules\Extensions** folders, right-click the *prefix***bmf_extension_workflow_sample** extension you have created, and choose **Generate extension code**. (*prefix* corresponds to the naming prefix for your project, for example, **G4_**.)

The extension boilerplate code is generated into a *prefix***bmf_extension_workflow_sample.cxx** C ++ file and a *prefix***bmf_extension_workflow_sample.hxx** header file. To see these files, open the project in the **Navigator** view and browse to the **src\server***library* directory.

> Note:
>
> You may need to right-click in the view and choose **Refresh** to see the files that were generated.

3.   Using the generate boilerplate code files, create the custom code. Following is an example of the *prefix***bmf_extension_workflow_sample.c** source file.

> Note:
>
> In the following example, change **G4_** to your project's naming prefix.

```
#include <ug_va_copy.h>
#include <itk/bmf.h>
#include <epm/epm.h>
#include <epm/epm_task_template_itk.h>

int getArgByName(const BMF_extension_arguments_t* const p, const int arg_cnt,
const char*
paramName)
    {
        int i = 0;
```

```
            for(; i < arg_cnt; i++)
            {
                    if(tc_strcmp(paramName, p[i].paramName) == 0)
                            return i;
            }

            return -1;
}

int G4_bmf_extension_workflow_sample(METHOD_message_t *msg, va_list args)
{
            int  ifail = ITK_ok;

            tag_t* new_item = NULL;
            tag_t* new_rev = NULL;

            tag_t item_tag = NULLTAG;
            tag_t rev_tag = NULLTAG;
            tag_t job_tag = NULLTAG;
            tag_t template_tag = NULLTAG;

            char* item_id = NULL;
            char* item_name = NULL;
            char* type_name = NULL;
            char* rev_id = NULL;

            char relproc[BMF_EXTENSION_STRGVAL_size_c + 1] = {'\0'};
            char job_desc[WSO_desc_size_c + 1] = {'\0'};
            char job_name[WSO_name_size_c + 1] = {'\0'};

            int index = 0;
            int paramCount = 0;
            int companyid = 0;
            int attachment_type = EPM_target_attachment;

            BMF_extension_arguments_t* input_args = NULL;

            /*******************/
            /*  Initialization  */
            /*******************/
            //Get the parameters from the ITEM_create_msg
            va_list largs;
            va_copy(largs, args);
            item_id = va_arg( largs, char *);
            item_name = va_arg(largs, char *);
            type_name = va_arg(largs, char *);
            rev_id = va_arg(largs, char *);
            new_item = va_arg(largs, tag_t *);
            new_rev = va_arg(largs, tag_t *);
            item_tag = *new_item;
            rev_tag = *new_rev;
            va_end(largs);

            // Extract the user arguments from the message
            ifail = BMF_get_user_params(msg, ¶mCount, &input_args);

            if(ifail == ITK_ok && paramCount >= 2)
            {
                    index = getArgByName(input_args, paramCount, "Release Process");
```

```
                        if(index != -1)
                                tc_strcpy(relproc, input_args[index].arg_val.str_value);

                        index = getArgByName(input_args, paramCount, "Company ID");

                        if(index != -1)
                                companyid = input_args[index].arg_val.int_value;

                        MEM_free(input_args);

                        if(relproc != NULL && rev_tag != NULLTAG)
                        {
                                sprintf( job_name, "%s/%s-%d Job", item_id, rev_id,
companyid);
                                sprintf( job_desc, "Auto initiate job for Item/
ItemRevision (%s/%s-%d)",
item_id, rev_id, companyid);

                                //Get the template tag.
                                ifail = EPM_find_template(relproc , 0, &template_tag);

                                if(ifail != ITK_ok)
                                        return ifail;

                                // Create the job, initiate it and attach it to the item
revision.
                                ifail = EPM_create_process(job_name, job_desc,
template_tag, 1, &rev_tag,
&attachment_type, &job_tag);

                                if(ifail != ITK_ok)
                                        return ifail;
                        }
                        else
                        {
                                // User supplied error processing.
                        }
                }

        return ifail;
    }
```

Following are some considerations when creating custom code:

Keep in mind:

- An extension should be expected to process one message.

- The message data and custom extension user data are independent of each other so it does not matter which is processed first. However, they should be processed together as a set.

- You are expected to know the data that exists for the message. This includes data types and the retrieval order so this data can be processed properly using **va_arg**.

- You are expected to know the user data that exists for the custom extension. This includes data types and whether each data element is mandatory or optional so that this data can be processed properly after the **BMF_get_user_params** function is used to retrieve this data.

- Several extensions can be compiled into the same library.

  - Copy all the implementation C files to the same directory.

  - Use the compile scripts (discussed below) and specify multiple targets to be compiled.

  - Execute the link script once to produce the library containing these extensions.

Workflow extension example: Execute the sample extension

1. Deploy the customization to a test server by choosing **BMIDE→Deploy Template** on the menu bar.

2. In the rich client, launch My Teamcenter.

3. Create a new item.

You see a TCM workflow process attached to the item revision when the item is created.

Working with user exits

The **User Exits** folder in the **Extensions** folder is for working with *user exits*, mechanisms for adding base action extension rules. User exits are places in the server where you can add additional behavior by attaching an extension.

> Note:
>
> User exit attachments, unlike operation extension attachments, cannot be attached at the level of child business objects. User exit attached extensions get defined and executed as callbacks only at a particular business object.

To work with user exits, right-click a user exit and choose **Open Extension Rule**. You can add actions on the right side of the editor. The **Base-Action** section is only shown for user exits operations.

You can make a user exit available for use when you define an extension rule. Right-click the **Extensions** folder, choose **New Extension Definition**, click the **Add** button to the right of the **Availability** table, and click the **Browse** button on the **Business Object Name** box. The user exits display on the list along with business objects.

The *Integration Toolkit Function Reference*, available on Support Center, documents users exits.

Basic concepts for data model customization

Data model customization framework

The data-model-based customization framework supports a single coherent mechanism for developing new functionality using C++. The framework:

- Defines business logic on business objects in the **Impl** class of the business object.

- Provides a consistent approach that can be used for internal Siemens Digital Industries Software development as well as for external customizations.

- Is scalable (delayed loading of DLLs).

- Exposes the business logic API in an object-oriented way (through interfaces).

- Improves memory performance.

Using the Business Modeler IDE, you can autogenerate the C++ plumbing code required to implement the business logic. This:
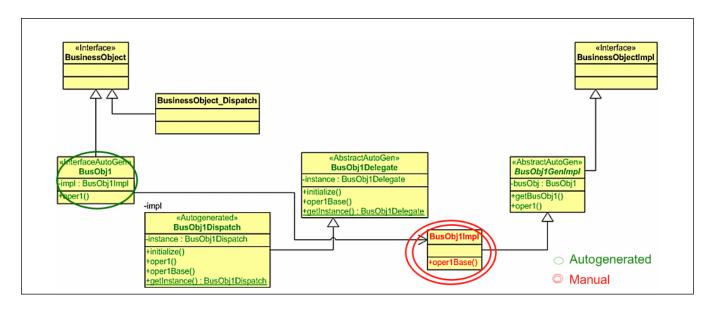
- Reduces implementation time.

- Increases productivity because you can spend more time in writing business logic.

- Enforces coding standards.

- Aids future enhancements.

Data model customization framework paradigm

The data model customization framework has the following paradigm.

- **BusinessObject** interface that defines the API for the business object. All callers work only with the interface.

- **Dispatch** class that implements the interface. It provides the overriding and extensibility capabilities.

- An abstract generated implementation class. This class provides getters and setters for the attributes on the business object and any other helper methods needed by the implementation.

- **Impl** class that is the actual implementation where the business logic is manually implemented by the developer.

- Delegate class that delegates invocations to the implementation class and breaks the direct dependency between the dispatch and implementation classes.

Framework interfaces

Class	Auto-generated	Purpose
Interface	Yes	C++ interface class. Defines published/unpublished API. All server caller code dependent on this business object works with only this interface.
Dispatch	Yes	Implements the interface. Provides the overriding and extensibility (pre/post) capabilities to the business object.
Generated Implementation (**GenImpl**)	Yes	C++ abstract Class. This class provides methods to get/set attribute values in the database and any other helper methods needed by the implementation.
Implementation (**Impl**)	No	Implementation class. The developer implements the core business logic for the business object in this class.
Delegate	Yes	C++ abstract class. This class breaks the direct dependency between the dispatch and implementation classes.

Note:

Code is autogenerated in the Business Modeler IDE by opening the **Advanced** perspective, right-clicking your project folder in the **Business Objects** view, and choosing **Generate Code→C++ classes**.

Data model customization framework tier structure

In data model framework customization, most of the code pieces are autogenerated, while you only need to write code in the implementation (**Impl**) file, as shown in the following figure.

First you create the business objects and operations, and their definitions are deployed to the database without writing any code. Then you generate code, creating boilerplate files that provide inheritance and overriding behavior to the business object. Finally, you write the custom logic for the operation in the stub provided in the **Impl** file.

Generated code

Server interaction with the framework

The data model customization framework supports a single coherent mechanism for developing new business functionality by defining business logic on business objects. This framework isolates business logic in one location: the **Impl** class, as shown in the following figure.

Basic tasks for data model customization

Enable data-model-based customization

To enable server customization, configure the Business Modeler IDE to generate C++ boilerplate code and write implementation code.

1. Install the Business Modeler IDE.

    In the **Solutions** panel of the Teamcenter Environment Manager (TEM), select **Business Modeler IDE**.

2. Create a project in the Business Modeler IDE.

    Choose **File**→**New**→**Project**, and in the **New Project** dialog box, choose **Business Modeler IDE**→**New Business Modeler IDE Template Project**.

    As you work through the wizard, ensure that you fill in the **Code Generation Information** dialog box and the **Build Configuration Information** dialog box.

3. Set up the coding environment in the **Advanced** perspective of the Business Modeler IDE.

    Create the following objects:

    - Release

      In the **Extensions** view, choose **Code Generation**→**Releases**, right-click the **Releases** folder and choose **New Release**.

    - Library

      In the **Extensions** view, choose **Code Generation**→**Libraries**, right-click the **Releases** folder and choose **New Library**.

    - External data types (if needed)

      In the **Extensions** view, open the **Code Generation**→**Data Types** folder, and right-click the **External Data Type** folder and choose **New External Data Type**.

After the environment is set up, you are ready to create C++ customizations.

Subclass and extend operations

The focus of server customization is defining business logic on business objects. To perform this kind of customization, you must use the Business Modeler IDE to create custom business objects and then place operations on those business objects.

1. Access the **Advanced** perspective by choosing **Window**→**Open Perspective**→**Other**→**Advanced**.

2. Create a custom business object.

   In the **Business Objects** view, right-click the business object you want to extend (for example, **Item**) and choose **New Business Object**.

3. Create an operation on a custom business object or property, or override an operation.

   In the **Business Objects** view, right-click the custom business object, choose **Open**, and in the resulting editor, click the **Operations** tab.

   Perform any of the following:

   • To add a business object operation, click the **Operations** folder and click the **Add** button.

   • To add a property operation, click a custom property in the **Property Operations** folder and click the **Add** button.

   • To override an operation, click the operation you want to override in the **Operations** or the **Property Operations** folder, and click the **Override** button. (The **Override** button is available if the operation is inherited from a parent business object, is marked that it can be overridden, and is not already overridden.)

4. Save the new business object and the operation changes.

   Choose **File**→**Save Data Model**, or click the **Save Data Model** button on the main toolbar.

5. Generate code.

   In the **Business Objects** view, right-click your project folder and choose **Generate Code**→**C++ classes**.

6. Write an implementation in the generated *business_object***Impl.cxx** file.

   Right-click in the **Navigator** view and choose **Refresh**. Add your implementation at the bottom of the **Source Folder***library\business_object***Impl.cxx** file where it contains the following text:

   ```
   ** Your Implementation **
   ```

7. Build the libraries containing server code.

    Choose **Project→Build Project**.

8. Package your template.

    Choose **File→New→Other**, and in the **New** dialog box, choose **Business Modeler IDE→Generate Software Package**. This packages your changes into a template that can be installed to a test server.

    > Note:
    >
    > You cannot use live update to distribute codeful customizations to a server. The Business Modeler IDE packages the built C++ library as part of its template packaging. The complete solution that includes the data model and the C++ run-time libraries are packaged together.

9. Install the template to a test server to verify the new functionality.

    After packaging, copy the resulting files from your *project***\output\packaging** directory to a directory on the test server where you want to install the template.

    Using the Teamcenter Environment Manager (TEM), in the **Features** panel, click the **Browse** button on the lower right side of the panel to browse to the directory where you have copied the template files.

Perform C++ coding

Perform C++ coding in the **C/C++** perspective in the Business Modeler IDE.

1. Access the **Advanced** perspective by choosing **Window→Open Perspective→Other→Advanced**.

2. After writing a new operation or overriding an operation, generate the boilerplate code.

    In the **Business Objects** view, right-click your project folder and choose **Generate Code→C++ classes**.

3. Switch to the **C/C++** perspective

    Choose **Window→Open Perspective→Other→C/C++**.

4. Open the code file.

    Right-click the **Source Folder***library\business_object***Impl.cxx** file in the **Navigator** view and select **Open** or **Open With**.

    The file opens in an editor.

5. Write the implementation.

Add your implementation at the bottom of the *business_object***Impl.cxx** file where it contains the following text:

```
** Your Implementation **
```

The implementation you write depends on what you want to accomplish:

- Create a new operation

  For example, you add an operation named **myOperation** to the **MyObject** business object. The business logic for the **myOperation** operation needs to be implemented in the **MyObjectImpl::myOperationBase** method:

  ```
  /**
  * This is a test operation.
  * @param testParam — Test Parameter
  * @return — Status. 0 if successful
  */
  int MyObjectImpl::myOperationBase( int testParam )
  {
      // Your Implementation
  }
  ```

- Override an operation

  For example, you override the **myOperation** operation in the **MySubObject** business object. The business logic for the override needs to be implemented in the **MySubObjectImpl::myOperationBase** method. It needs to call **MySubObjectGenImpl::super_myOperationBase** to invoke the parent implementation:

  ```
  /**
  * This is a test operation.
  * @param testParam — Test Parameter
  * @return — Status. 0 if successful
  */
  int MySubObjectImpl::myOperationBase( int testParam )
  {
       // Your Implementation


     // Invoke super implementation
     stat = MySubObjectGenImpl::super_myOperationBase( int
  testParam );
  ```

```
        // Your Implementation

    }
```

> **Note:**
>
> You should not call **MyObjectImpl::myOperationBase** for invoking parent implementation.

- Call another operation

    For example, you want to call the **myOperation2** operation from the **myOperationBase** method. The **myOperationBase** method should invoke the **myOperation2** operation on itself:

    ```
    /**
    * This is a test operation.
    * @param testParam — Test Parameter
    * @return — Status. 0 if successful
    */
    int MyObjectImpl::myOperationBase( int testParam )
    {
        // Your Implementation


      // Invoke another operation
      // Generate in MyObjectGenImpl
      myOperation2( int testParam );

        // Your Implementation

    }
    ```

    The **myOperation2** operation is auto-generated in the **MyObjectGenImpl** implementation.

    > **Note:**
    >
    > You should not call **MyObjectImpl::myOperation2Base**.

Working with the create operation

The create operation

The **create** operation is an interface operation. You can view this operation and its child operations in the Business Modeler IDE on the **Operation** tab for all business objects.

The **create** operation in turn calls a protected method, **createBase**, that is implemented as an orchestration method and is not overridable, as shown in the following example:

```
int POM_objectImpl::createBase( Teamcenter::CreateInput * creInput )
{
    try
    {
        ResultStatus stat;
        stat = finalizeCreateInput(creInput);
        stat = validateCreateInput(creInput);
        stat = createInstance(creInput);
        stat = setPropertiesFromCreateInput(creInput);
        stat = createPost(creInput);
    }

}
```

The child operations that are marked in bold in the example can be overridden at a child business object:

- **finalizeCreateInput** operation

  Override this operation to implement any special business logic code for handling computed properties. Use it to:

  - Compute property values if not defined.

  - Assign a computed value to the **CreateInput** object using generic get/set APIs.

  For example, the **item_id** value is computed using the existing **USER_new_item_id** user exit in the following method:

  ```
  ItemImpl::finalizeCreateInputBase (Teamcenter::CreateInput * creInput )
  ```

  > Note:
  >
  > Whereas the **USER_new_item_id** user exit is used to create the ID for a single item, the **USER_new_item_ids** user exit is used for bulk creation of the IDs for all the associated objects that are created at the same time as the item (revisions, forms, and so on).

- **validateCreateInput** operation

  Override this operation to implement any special business logic code for validating the **CreateInput**. You can use it to perform uniqueness validation, for example, the **item_id** input is unique.

- **setPropertiesFromCreateInput** operation

Override this operation to implement any special business logic code for populating the POM instance. Use it to:

- Populate input values onto the corresponding attributes of the POM instance. (This is implemented generically at the **POM_object** level.)

- Populate or initialize special attributes in terms of special business logic. (This should be avoided.)

- **createPost** operation

  Override this operation to implement any special business logic code for creating compounded objects.

  - Create compound objects or secondary objects.

  - Assign compound or secondary objects to the primary object through references or GRM relations.

  For example, the master form and item revision on an **Item** business object are created in:

  ```
  ItemImpl::createPostBase( Teamcenter::CreateInput * creInput )
  ```

  > Note:
  >
  > The create operation logically creates objects and then the save operation commits the changes to the database. This pattern must be followed whenever the object creation is initiated. Following this pattern requires that any postactions on the create operation should not assume the existence of the objects in the database. If database interactions like queries or saving to the database against the newly created objects are needed in the postactions, they must be attached as a postaction to the save message and not the create message. The save message is initiated following the create or modify operation. To distinguish between the save types, use the following C API:
  >
  > ```
  > TCTYPE_ask_save_operation_context
  > ```

Implement the create operation on a new business object

1. Use the **Operation Descriptor** tab on the new business object to configure its creation properties.

2. Override the child operations on the **create** operation in the **Operations** tab for the new business object.

3. Generate the C++ code in the **Advanced** perspective by right-clicking your project folder in the **Business Objects** view and choosing **Generate Code→C++ classes**.

4. Implement custom create business logic in the generated **Impl** file.

Single API signature for the create operation

2-91
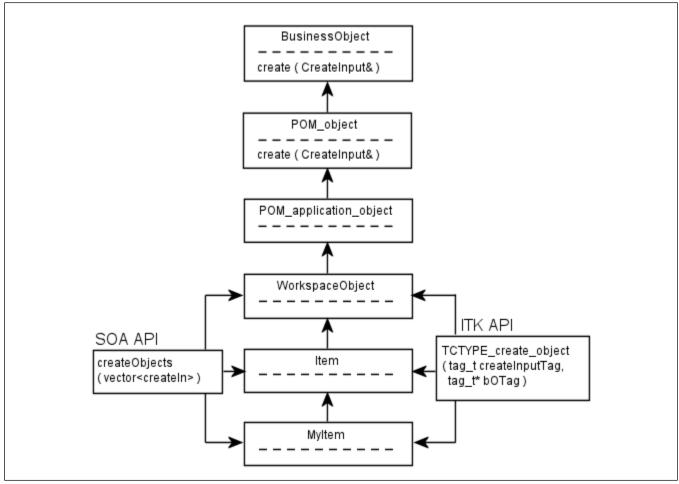
The data model customization framework uses a single C++ API for creating any business object: **CreateInput**. It works with the **creInput** object to hold name/value pairs that are used for the business object creation.

The single API structure:

- Enables each business object to have its own set of attributes that can be passed into the create operation.

- Allows a business object to override the parent create operation to add additional validations or to handle additional processing.

- Handles additional properties added to a COTS business object by an industry vertical solution or a customer.

- Handles sub-business objects in a codeless manner (no code generation and writing code) when they are added by the core model, an industry vertical solution, or a customer.

The metadata for the **CreateInput** API is different for different business objects and is configured on the **Operation Descriptor** tab for the business object in the Business Modeler IDE. This metadata is inherited and can be overridden down the business object hierarchy. You can add additional attributes, and clients can query this metadata to provide the appropriate create user interface.

Single API signature for the create operation

Calling generic creation C++ API at the server side

To call the generic creation **CreateInput** C++ API at the server side, follow these steps:

1. Construct a **CreateInput** object.

2. Populate the **CreateInput** object with the input data, using generic property set (C++) APIs.

3. Construct **CreateInput** objects for compounded objects and populate and assign them to the **CreateInput** object of the primary object.

4. Call generic creation by passing in the **CreateInput** object.

5. Save the primary object.

6. Delete all the **CreateInput** objects.

The resulting code might appear similar to the following sample:

```
Int MyBoImpl::myOpBase( … )
{
    //your implementation

    //Calling Generic Creation (C++)
     Teamcenter::CreateInput* pItemCreInput = dynamic_cast<Teamcenter::CreateInput*>
(Teamcenter::BusinessObjectRegistry::instance().createInputObject
("Item", OPERATIONINPUT_CREATE));
    pItemCreInput->setString("item_id", item_id, false);

    Teamcenter::CreateInput* pRevisionCreInput = dynamic_cast<Teamcenter::CreateInput*>
(Teamcenter::BusinessObjectRegistry::instance().createInputObject("ItemRevision",
OPERATIONINPUT_CREATE));
pRevisionCreInput->setString("item_revision_id", rev_id, false);

    tag_t item_rev_createinput_tag =
((Teamcenter::BusinessObject*)pRevisionCreInput)->getTag();
    pItemCreInput->setTag("revision", item_rev_createinput_tag, false);

    pItem = dynamic_cast<Teamcenter::Item
*>(Teamcenter::BusinessObjectRegistry::instance().
createBusinessObject(pItemCreInput));

}
```

Calling generic creation Teamcenter Services API from the client

1. Construct a **CreateInput** object and populate the input into the appropriate property value map.

2. Construct a **CreateInput** object for the compounded objects and populate and assign them into the compound object map.

3. Make a single services invocation to create the business object.

The resulting code may appear similar to the following example:

```
CreateIn createIn = new CreateIn();
   createIn.data.boName = "Item";
   createIn.data.stringProps.put("item_id", "Item001");

CreateInput revCreateIn = new CreateInput();
  revCreateIn.boName = "ItemRevision";
  revCreateIn.stringProps.put("item_revision_id","ItemRevSOATest");

CreateInput itemMasterCreateIn = new CreateInput();
  itemMasterCreateIn.boName = "Item Master";
  itemMasterCreateIn.stringProps.put("project_id", "ItemProjectID");

CreateInput itemRevMasterCreateIn = new CreateInput();
  itemRevMasterCreateIn.boName = "ItemRevision Master";
  itemRevMasterCreateIn.stringProps.put("project_id", "ItemRevProjID");

revCreateIn.compoundCreateInput.put("IMAN_master_form_rev", new CreateInput[]
{itemRevMasterCreateIn});
  createIn.data.compoundCreateInput.put("revision", new CreateInput[]{revCreateIn});
  createIn.data.compoundCreateInput.put("IMAN_master_form", new CreateInput[]
```

```
   {itemMasterCreateIn});

// Call the createObjects operation
   CreateResponse resp = dataMgtService.createObjects(new CreateIn[]{createIn});
```

> **Note:**
>
> Whereas the **createObjects** service operation is used to create a single object, the
> **bulkCreateObjects** service operation is used for bulk creation of all the associated objects when
> you create an object with associated objects, such as an item business object.

Call Teamcenter C++ API

You can call the standard Teamcenter C++ API in your C++ implementation code.

Reference information about Teamcenter C++ APIs can be found in the *C++ API Reference*. (The
Integration Toolkit Function Reference is available on Support Center.)

When you reference Teamcenter C++ API, typically, the customization code starts with a **tag_t** tag, then
constructs a business object reference with the **itemTag** tag, and finally calls the C++ API, as in the
following example:

```
#include <metaframework/BusinessObjectRef.hxx>
#include <fclasses/ifail.hxx>
#include <tccore/Item.hxx>

typedef unsigned int tag_t;
#define ITK_ok  0
#define EMPTY_ITEM_ID 101; // error code to be defined
#define UNKNOWN_ERROR  201; // error code to be defined

int setWsoObjectDescAndCheckItemId (tag_t itemTag, const std::string& description)
{
    int ifail = ITK_ok;

    // create a smart pointer of Item Type from object tag
    BusinessObjectRef<Teamcenter::Item> item(itemTag);
    try
    {
        if (description.empty())
        {

            // execute Teamcenter::WorkspaceObject::setObject_desc method
            // Note that WorkspaceObject is the parent BusinessObject of Item
            if ((ifail = item->setObject_desc(description, true)) != ITK_ok )
            {
                return ifail;
            }
        }
        else
        {
            // execute Teamcenter::WorkspaceObject::setObject_desc method
            if (( ifail=item->setObject_desc(description))!= ITK_ok)
            {
```

```
                return ifail;
            }
        }
        std::string itemId;
        bool isNull = false;
        // execute Teamcenter::Item::getItem_id method
        if ((ifail = item->getItem_id( itemId, isNull))!= ITK_ok )
        {
            return ifail;
        }
        if ( isNull )
        {
            return EMPTY_ITEM_ID; // empty item_id
        }

    }
    catch ( IFail &ex )
    {
        return ex;
    }
    catch (...)
    {
        printf ("\n uncepted exception captured; please report errors. \n");
        return UNKNOWN_ERROR;
    }
    return ITK_ok;
```

3. Teamcenter Services customization

Introduction to Teamcenter Services customization

Teamcenter Services are a collection of Teamcenter operations. Teamcenter Services are used to integrate your company's applications to Teamcenter.

Services are organized as follows:

- Operations

  Define discrete functions, such as create, checkin, checkout, and so on. Operations are used for all internal Teamcenter actions and are called by external clients to connect to the server.

- Services

  Organize operations into groups having to do with a specific area.

- Libraries

  Place similar services together.

Teamcenter ships with service libraries and interfaces to build applications in Java, C++, and C# (.NET), allowing you to choose the method that best fits with your environment. Teamcenter also ships with WS-I compliant WSDL files for all operations. To see these libraries, see the **soa_client.zip** file on the installation source.

A *Services Reference* is available in the Teamcenter HTML help.

> Note:
>
> It is not available in the PDF help.

Enable Teamcenter Services customization

To enable Teamcenter Services customization, configure the Business Modeler IDE to create services and write the implementation code.

1. Install the Business Modeler IDE.

   In the **Solutions** panel of the Teamcenter Environment Manager (TEM), select **Business Modeler IDE**.

During installation, you are asked to enter the location of a Java Development Kit (JDK) on your system. After installation, verify that the JDK is set in the *install-location***\bmide\client\bmide.bat** file, for example:

```
set JRE_HOME=C:\Program Files (x86)\Java\jre7
set JAVA_HOME=C:\Program Files (x86)\Java\jdk1.7.0
set JDK_HOME=C:\Program Files (x86)\Java\jdk1.7.0
```

2. Create a project in the Business Modeler IDE.

   Choose **File→New→Project**, and in the **New Project** dialog box, choose **Business Modeler IDE→New Business Modeler IDE Template Project**.

   As you work through the wizard, ensure that you fill in the **Code Generation Information** dialog box and the **Build Configuration Information** dialog box.

3. Set up the services creation environment in the Business Modeler IDE.

   Create the following objects:

   - Release

     In the **Extensions** view of the **Advanced** perspective, choose **Code Generation→Releases**, right-click the **Releases** folder, and choose **New Release**.

     Ensure that you select the **Set as Current Release** check box to indicate that this release is the active one to use for your new services. In the **Service Version** box, type the version of the services you plan to create in the release using format _*YYYY_MM*, for example, **_2011_10**. You *must* fill in the **Service Version** box if you plan to create services for this release.

   - Service library

     In the **Extensions** view, choose **Code Generation→Services**, right-click the **Services** folder and choose **New Service Library**.

After the environment is set up, you are ready to create services.

Basic tasks for Teamcenter Services customization

Add Teamcenter Services

Use the Business Modeler IDE to add Teamcenter Services and operations.

1. Add a service.

In the **Extensions** view of the **Advanced** perspective, expand the **Code Generation→Services** folders, right-click the service library in which you want to create the new service, and choose **New Service**.

2. Add service data types (if needed). Service data types are used by service operations as return data types and can also be used as parameters to operations.

   Right-click the service, choose **Open**, click the **Data Types** tab, and click the **Add** button.

3. Add service operations.

   Right-click the service, choose **Open**, click the **Operations** tab, and click the **Add** button.

4. Generate service artifacts.

   Right-click the **Code Generation→Services** folder and choose **Generate Code→Service Artifacts**. To generate code only for a particular service library, select only the service library.

5. Write implementations of the service operations.

   To see the generated files, right-click in the **Navigator** view and choose **Refresh**. By default, the generated service artifacts are placed in the **Source Folder**_service-library_ folders.

   Write implementations in the generated _service_**impl.cxx** files where it contains the text: **TODO implement operation**.

6. Build the libraries containing server code.

   In the **Navigator** view, right-click the project and choose **Build Configurations→Build→All**.

7. Package your template.

   Choose **File→New→Other**, and in the **New** dialog box, choose **Business Modeler IDE→Generate Software Package**. This packages your changes into a template that can be installed to a test server.

   > Note:
   >
   > You cannot use live update to distribute codeful customizations to a server.

8. Install the template to a test server to verify the new functionality.

   After packaging, copy the resulting files from your _project_**\output\packaging** directory to a directory on the test server where you want to install the template.

Using the Teamcenter Environment Manager (TEM), in the **Features** panel, click the **Browse** button on the lower right side of the panel to browse to the directory where you have copied the template files.

Connect a client to Teamcenter Services

Use Teamcenter Services samples as a guideline to create your own clients to connect to the server-side Teamcenter Services.

You can use the services sample client applications as a guideline to configure your own clients. Sample clients are provided in the **soa_client.zip** file on the Teamcenter software distribution image. When you unzip the **soa_client.zip** file, the **cpp**, **java**, and **net** folders are created, and under each is a samples folder that contains sample clients. For more information about how to configure your own client, look at the source files provided with the samples.

Services

Process for creating services in the Business Modeler IDE

You can create your own Teamcenter services and service operations using the Business Modeler IDE.

A *service* is a collection of service operations that all contribute to the same area of functionality. A *service operation* is a function, such as create, checkin, checkout, and so on. Service operations are used for all internal Teamcenter actions, and are called by external clients to connect to the server.

When you use the Business Modeler IDE to create your own services and service operations, follow this process:

1. **Set up code generation**.

2. **Add a service library**.

3. **Add a service**.

4. **Add service data types**.

5. **Add service operations**.

6. **Generate service artifacts**.

7. **Write implementations of the service operations**.

8. **Build server and client libraries**.

Teamcenter ships with service libraries and interfaces to build applications in Java, C++, and C# (.NET), allowing you to choose the method that best fits with your environment. Teamcenter services also ships with WS-I compliant WSDL files for all operations. To see these libraries, see the **soa_client.zip** file on the installation source.

Add a service library
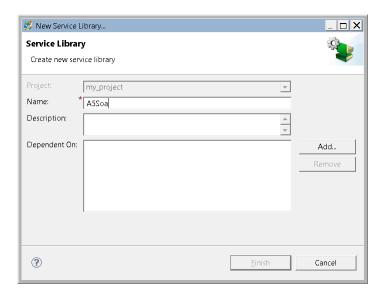
A *service library* is a collection of files used by a service and its operations that includes programs, helper code, and data. Services are grouped under service libraries. You must create a service library before you create a service.

1. Right-click the project and choose **Organize→Set active extension file**.

    You can create your own XML file, such as **services.xml**. In the **Project Files** folder, right-click the **extensions** folder and choose **Organize→Add new extension file**.

2. Choose one of these methods:

    -   On the menu bar, choose **BMIDE→New Model Element**, type **Service Library** in the **Wizards** box, and click **Next**.

    -   Open the **Extensions\Code Generation** folders, right-click the **Services** folder, and choose **New Service Library**.

    The New Service Library wizard runs.

3. Perform the following steps in the **Service Library** dialog box:

    a. In the **Name** box, type the name you want to assign to the new service library.

When you name a new data model object, a prefix from the template is automatically affixed to the name to designate the object as belonging to your organization. The letters **Soa** are added to the prefix for service library names, for example, **A4_Soa**.

b.　In the **Description** box, type a description of the new service library.

c.　If this library requires other libraries in order to function, click the **Add** button to the right of the **Dependent On** box to select other libraries.

> Design Intent:
>
> This option updates the build files with linking information.

d.　Click **Finish**.

The new library appears under the **Services** folder.

4.　To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

5.　**Create services** to go into the library. Right-click the library and choose **New Service**.

Add a service

A *service* is a collection of Teamcenter actions (service operations) that all contribute to the same area of functionality. Before you can add a service, you must create a service library to place the service into.

Teamcenter ships with its own set of services and service operations to which you can connect your own client. See the **soa_client.zip** file on the installation source.

1.　Expand the **Extensions\Code Generation\Services** folders.

2.　Under the **Services** folder, select a **service library** to hold the new service, or create a new service library to hold the new service.

3.　Right-click the custom service library in which you want to create the new service, and choose **New Service**.
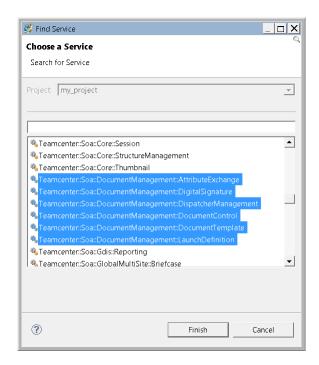
The New Service wizard runs.

4. Perform the following steps in the **Service** dialog box:

    a. The **Template** box displays the project to which this new service is added.

    b. In the **Name** box, type the name you want to assign to the new service.

    c. Click the **Description** button to type a description of the new service.

    d. Click the **Dependencies** button to select services that the new service is dependent upon. You can use the Ctrl or Shift keys to select multiple services.

    > Design Intent:
    >
    > This is a documentation option for your reference. It has no impact on build files or linking.

e.   Click **Finish**.

The new service appears under the service library in the **Services** folder.

5.   To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 🖫 on the main toolbar.

6.   Create **data types** and **operations** for the service. Right-click the service, choose **Open**, and click the **Data Types** tab or the **Operations** tab.

Service data types

Introduction to service data types

Service data types are used by service operations as return data types and can also be used as parameters to operations. When you **create a service operation**, service data types are shown as **Generated** data types in the **Choose a Data Type Object** dialog box. The data types on a service library are defined for use only by that library and are not shared with other libraries.

To create a service data type, open a service, click the **Data Types** tab, and click the **Add** button. You can create the following kinds of service data types:

• Structure

Contains a set of types. Corresponds to a C++ **struct** data type.

• Map

Links one type to another. Corresponds to a C++ **typedef** data type.

• Enumeration

Contains a list of possible values. Corresponds to a C++ **enum** data type.
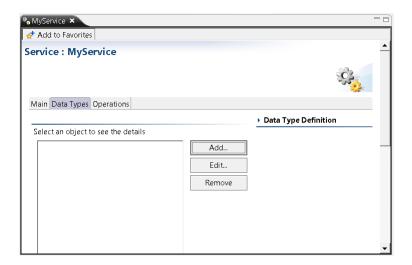
Add a structure data type

A *structure data type* is a kind of service data type that contains a set of types. It corresponds to a C++ **struct** data type. Service structures can be composed of many types: primitive types, string, enumerations, maps, business objects, structures (for example, **ServiceData**), as well as vectors of these (except for **ServiceData**).

> Note:
>
> **Service data types** are used only by the service library for which they are created.

1. Ensure that you have set the proper active release for the data type. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2. Expand the **Code Generation\Services** folders.

3. Under the **Services** folder, open a service library and select the service to hold the new data type. You must **create a service library** and **service** before you create a service data type.

4. Right-click the service in which you want to create the new data type, choose **Open**, and click the **Data Types** tab.

5. Click the **Add** button on the **Data Types** tab.

   The Complex Data Type wizard runs.

6. Select **Structure** and click **Next**.

   The **Structure** dialog box is displayed.

7. Perform the following steps in the **Structure** dialog box:

   a. The **Template** box defaults to the already-selected project.

   b. In the **Name** box, type the name you want to assign to the new data type.

   c. Select the **Published** check box to make the data type available for use by client applications. Clear the box to unpublish the type, and to put it into an **Internal** namespace. Unpublished types can only be used by unpublished operations (published can be used by either).

   d. Click the **Add** button to the right of the **Data Type Elements** table to add elements to the structure data type.

      The **Structure Element** dialog box is displayed.

Perform the following steps in the **Structure Element** dialog box:

    A.   In the **Name** box, type a name for the new element.

    B.   Click the **Browse** button on the **Type** box to select the data type object to use for the element.

> Note:
>
> When defining service types or operations, you can only reference structure data types from a previous release or only map data types or enumeration data types from the same release.

    C.   Click the **Edit** button to type a description for the new element.

    D.   Click **Finish**.

    e.   Click the **Add** button again to add more elements to the structure data type. The **Structure Elements** table shows a preview of the new data type.

    f.   Click the **Description** button to type a description for the new structure.

    g.   Click **Finish**.

           The new data type displays on the **Data Types** tab. To see the characteristics of the data type, select it and click **Data Type Definition** on the right side of the editor.

8.   To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 🖫 on the main toolbar.

9. Write a service operation that uses the new data type. When you **create a service operation**, service data types are shown as **Generated** data types in the **Choose a Data Type Object** dialog box.
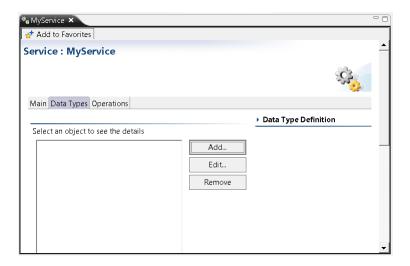
Add a map data type

A *map data type* is a kind of service data type that links one type to another. It corresponds to a C++ **typedef** data type of the **std::map** form. A map data type consists of two components, a key type and a value type.

> Note:
>
> **Service data types** are used only by the service for which they are created.

1. Ensure that you have set the proper active release for the data type. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2. Expand the project and the **Code Generation\Services** folders.

3. Under the **Services** folder, open a service library and select the service to hold the new data type. You must create a **service library** and **service** before you create a service data type.

4. Right-click the service in which you want to create the new data type, choose **Open**, and click the **Data Types** tab.

5. Click the **Add** button on the **Data Types** tab.
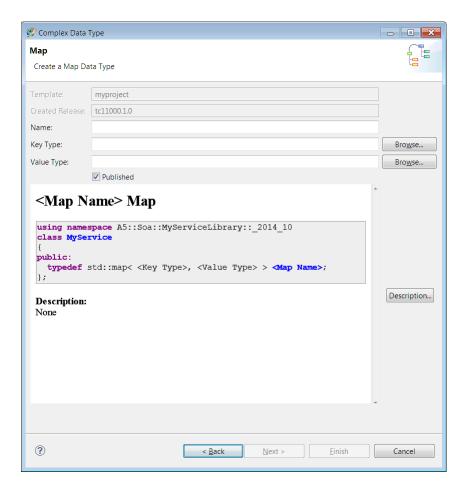
    The Complex Data Type wizard runs.

6. Select **Map** and click **Next**.

The **Map** dialog box is displayed.

7.   Perform the following steps in the **Map** dialog box:

a.   In the **Name** box, type a name for the data type.

b.   Click **Browse** to the right of the **Key Type** box to select the key to use for this map.

> Note:
>
> When defining service types or operations, you can only reference structure data types from a previous release or only map data types or enumeration data types from the same release.

c.   Click the **Browse** button to the right of the **Value Type** box to select the value to use for this map.

d.   Select the **Published** check box to make the data type available for use by client applications. Clear the box to unpublish the type, and to put it into an **Internal** namespace. Unpublished types can only be used by unpublished operations (published can be used by either).

e. Click the **Description** button to **type a description** for the new data type.

f. Click **Finish**.

The new data type displays on the **Data Types** tab. To see the characteristics of the data type, select it and click **Data Type Definition** on the right side of the editor.

8. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

9. Write a service operation that uses the new data type. When you **create a service operation**, service data types are shown as **Generated** data types in the **Choose a Data Type Object** dialog box.
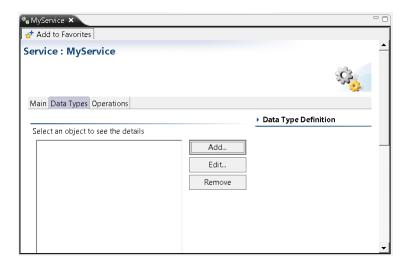
Add an enumeration data type

An *enumeration data type* is a kind of service data type that contains a list of possible values. It corresponds to a C++ **enum** data type. An enumeration data type consists of string names known as enumerators.

> Note:
>
> **Service data types** are used only by the service for which they are created.

1. Ensure that you have set the proper active release for the data type. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2. Expand the **Code Generation\Services** folders.

3. Under the **Services** folder, open a service library and select the service to hold the new data type. You must create a **service library** and **service** before you create a service data type.

4. Right-click the service in which you want to create the new data type, choose **Open**, and click the **Data Types** tab.

5.   Click the **Add** button on the **Data Types** tab.

The Complex Data Type wizard runs.

6.   Select **Enumeration** and click **Next**.

The **Enumeration** dialog box is displayed.

7. Perform the following steps in the **Enumeration** dialog box:

    a. In the **Name** box, type the name you want to assign to the new data type.

    b. Select the **Published** check box to make the data type available for use by client applications. Clear the box to unpublish the type, and to put it into an **Internal** namespace. Unpublished types can only be used by unpublished operations (published can be used by either).

    c. Click the **Add** button to the right of the **Enumerators** box to add enumerators to the data type.

        The New Enumeration Literal wizard runs. Type a value in the **Value** box and click **Finish**.

    d. Click the **Add** button again to add more enumerators to the data type, or click the **Remove** button to remove enumerators.

    e. Click the **Description** button to type a description for the new data type.

    f. Click **Finish**.

The new data type displays on the **Data Types** tab. To see the characteristics of the data type, select it and click **Data Type Definition** on the right side of the editor.

8. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

9. Write a service operation that uses the new data type. When you **create a service operation**, service data types are shown as **Generated** data types in the **Choose a Data Type Object** dialog box.
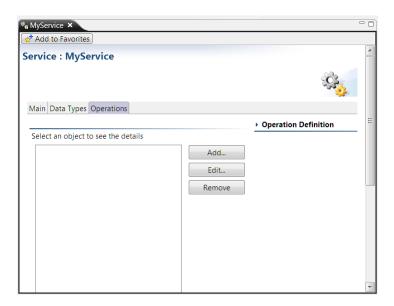
Add a service operation

A *service operation* is a function, such as create, checkin, checkout, and so on. Group service operations under a service to create a collection of operations that all contribute to the same area of functionality. You must create a **service library** and a **service** before you can create a service operation.

> Note:
>
> Teamcenter ships with its own set of services and service operations to which you can connect your own client. See the **soa_client.zip** file on the installation source.
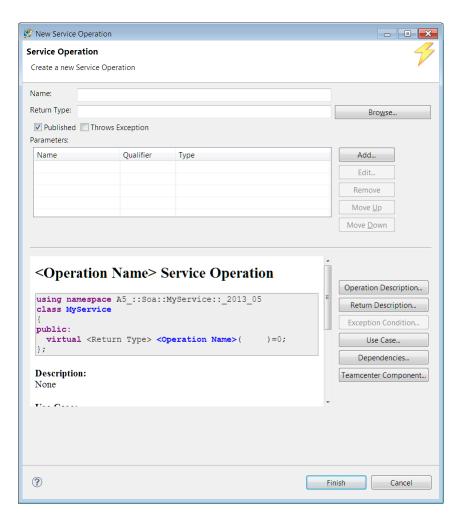
1. Ensure that you have set the proper active release for the operation. Open the **Extensions\Code Generation\Releases** folders, right-click the release, and choose **Organize→Set as active Release**. A green arrow in the release symbol indicates it is the active release.

2. Expand the **Code Generation\Services** folders.

3. Under the **Services** folder, open a service library and select the service to hold the new service operation. You must create a service library and service before you create a service operation.

4. Right-click the service in which you want to create the new service operation, choose **Open**, and click the **Operations** tab.

5. Click the **Add** button on the **Operations** tab.

The New Service Operation wizard runs.

6. Perform the following steps in the **Service Operation** dialog box:

   a. In the **Name** box, type the name you want to assign to the new service operation.

      The operation name combined with its parameters must be unique within this service. An operation name can only be reused if the duplicate name is in a different version.

   b. Click the **Browse** button to the right of the **Return Type** box to select the data type to be used to return data:

      • **bool**

        Returns a Boolean value (true or false). The **bool** data type is a **primitive data type**.

      • **std::string**

        String from the standard namespace. The **std::string** data type is an **external data type**.

      • **Teamcenter::Soa::Server::ServiceData**

        Returns model objects, partial errors, and events to the client application. If you select the **ServiceData** type, include it only in the top-level structure that a service is returning. The **ServiceData** data type is an external data type.

      You can also create your own structure service data types and select them in this dialog box.

      > Note:
      >
      > When defining service types or operations, you can reference only structure data types from a previous version, or only map data types or enumeration data types from the same version.
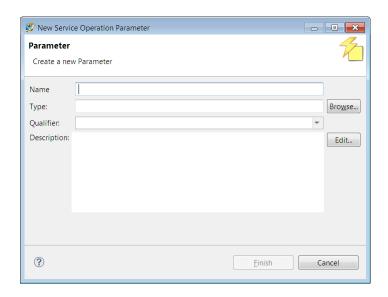
   c. Select the **Published** check box to make the operation available for use by client applications. Clear the box to unpublish the operation and place it into an **Internal** namespace.

   d. Select the **Throws Exception** check box if a return error causes an exception to be thrown.

      This enables the **Exception Condition** button.

   e. To set parameters on the service operation, click the **Add** button to the right of the **Parameters** table.

      The New Service Operation Parameter wizard runs.

Perform the following steps in the **Parameter** dialog box:

A.   In the **Name** box, type a name for the parameter.

B.   Click the **Browse** button to the right of the **Type** box to select a data type to hold the parameter.

In the **Choose a Data Type Object** dialog box, you can select the following data types to filter out:

- **Primitive**

  Generic data types. Only **boolean**, **double**, **float**, and **int** are available for services.

- **External**

  Standard data types. Only those listed are available for services; they are defined by Teamcenter.

- **Template**

  Data types that allow code to be written without consideration of the data type with which it is eventually used. Only **vector** is available for services.

- **Interface**

  Data types for returning business objects. Only business objects with their own implementation are selectable. For business objects without their own implementation, select the closest parent instead.

- **Generated**

  Service data types, such as structure, map, and enumeration data types. Only the data types set up for this service are shown as being available for this service. However, you can type in the name of a data type that is in another service as long as it is in the same service library.

  > Note:
  >
  > When defining service types or operations, you can only reference structure data types from a previous release or only map data types or enumeration data types from the same release.

C. Click the arrow in the **Qualifier** box to select the qualifier for the return value. Only the valid qualifiers display depending on the type of parameter selected.

- **&**

  Reference variable.

- *****

  Pointer variable.

- ******

  Double pointer. (Pointer to a pointer.)

- **[]**

  Array container.

D. Click the **Edit** button to type a complete description of this input parameter.

  Follow these best practices:

- Specify what this argument represents and how it is used by the operation.

- Specify if this parameter is optional. For an optional argument, describe the behavior if the data is not passed for that argument.

- Describe the result if a specific value is passed to this argument.

- Describe how data is to be serialized or parsed if the parameter type hides multiple property types behind a string.

- Avoid specifying just the type in the description. In case of hash maps, document the type of the map key and values.

- Use complete sentences.

- Use correct formatting with fixed and bold text where appropriate.

- Use correct Teamcenter terminology.

- Define acronyms before using them.

E. Click **Finish**.

The **Parameters** table displays the new parameter.

f. Click the **Operation Description** box to type a complete description of the functionality exposed through the service operation.

Follow these best practices:

- Describe what this operation does. Explain more than simply stating the method name.

- Make the description complete in its usefulness. Keep in mind the client application developer while writing the content.

- Whenever appropriate, describe how each argument works with other arguments and gets related to each other when this operation completes.

- Use correct formatting with fixed and bold text where appropriate.

- Use correct Teamcenter terminology.

- Define acronyms before using them.

g. Click the **Return Description** button to type a complete description of what the service operation returns. Follow these best practices:

- Describe what the output represents and provide high-level details of the output data. Do not specify only the type of service data returned.

- Describe any partial errors returned.

- Specify returned objects that are created, updated, or deleted as part of service data.

- Use complete sentences.

- Use correct formatting with fixed and bold text where appropriate.

- Use correct Teamcenter terminology.

- Define acronyms before using them.

h. Click the **Exception Condition** button to type a complete description of conditions that must be met for the operation to throw an exception.

This box is enabled when the **Throws Exception** check box is selected.

i. Click the **Use Case** button to describe how the user interacts with this operation to accomplish the operation's goal. Follow these best practices:

- Document when and why this operation can be consumed.

- Describe how operations interrelate to satisfy the use case (if there is interrelation between operations).

- Use complete sentences.

- Specify all possible use cases.

- Use correct formatting with fixed and bold text where appropriate.

- Use correct Teamcenter terminology.

- Define acronyms before using them.

j. Click the **Dependencies** button to select the other operations that are used in conjunction with this operation to accomplish the goal.

> Design Intent:
>
> This is a documentation option for your reference. It has no impact on build files or linking.

k. Click the **Teamcenter Component** button to select the component you want to tag this operation as belonging to. **Teamcenter Component** objects identify Teamcenter modules that have a specific business purpose.

> Tip:
>
> Create your own custom components to use for tagging your service operations. To create your own custom components, right-click **Teamcenter Component** in the

> **Extensions** folder, choose **New Teamcenter Component**, and type the details for the name, display name, and description. In the **Name** box, you can either choose your template name or choose a name to help you easily organize and identify your set of service operations. After the component is created, it appears in the list of available components.

l. The preview pane shows how the operation information appears in the service API header.
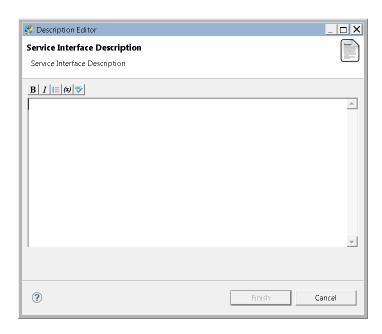
Click **Finish**.

The new service operation displays on the **Operations** tab. To see the characteristics of the operation, select it and click **Operation Definition** on the right side of the editor.

7. To save the changes to the data model, choose **BMIDE→Save Data Model**, or click the **Save Data Model** button 💾 on the main toolbar.

8. **Generate service artifacts**. Right-click the service containing the service operation and choose **Generate Code→Service Artifacts**.

Formatting text with the Description Editor dialog box

Use the **Description Editor** dialog box to format description text. This editor is displayed while you are creating or modifying a service, service operation, or service data type.

Click the buttons at the top of the editor to format the text:

- **Bold**

Applies a bold font to the selected text. You can also apply bold text by pressing the CTRL+B keys.

- **Italics**

  Applies italicized font to the selected text. You can also apply italicized font by pressing the CTRL+I keys.

- **Bullets**

  Applies bullets to the selected items. You can also apply bullets by pressing the CTRL+P keys.

- **Variable**

  Applies a fixed-width font (such as Courier) to the selected text to designate that the text is a variable. You can also apply the variable format by pressing the CTRL+O keys.

- **Check Spelling**

  Performs a spell check for the text in the **Description Editor** dialog box. You can also perform a spell check by pressing the CTRL+S keys.

Generate service artifacts
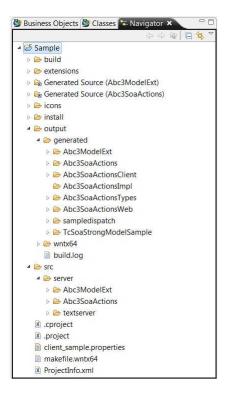
After you create operations, you must generate the service artifacts (source code files) from the operations and write your implementations in an outline file. Using the **Generate Code→Service Artifacts** menu command, you can generate the service source code files.

1. **Write operations for a service**.

2. Ensure that the code generation environment is set up in the project the way you want it.

   a. Right-click the project folder and choose **Properties**.

   b. Choose **Teamcenter→Build Configuration** and **Teamcenter→Code Generation**.

3. To generate code for all services, open the **Extensions\Code Generation** folders, right-click the **Services** folder, and choose **Generate Code→Service Artifacts**.

   In addition to generating source code files for the customization, the generate process also creates makefiles for C++ artifacts, Ant **build.xml** files for Java artifacts, and .csproj files for .NET artifacts, which are used to compile the source code. There is a platform-specific makefile created in the project's root folder (for example, **makefile.wntx64**) and there are also a number of platform-neutral build files created under the build folder for each library that is defined in the template. All of these build files may be persisted and are only updated by the Business Modeler IDE when something changes (such as a new library is defined or build options are changed). The root

makefile (for example, **makefile.wntx64**) is used to compile all library source files for the template. This includes the autogenerated code and your hand-written implementation code.

4. All generated files are placed in the **output/generated** folder, with a subfolder for each library. The implementation stub files are placed under the **src/server** folder, with a subfolder for each library.

5. **Write implementations** in the generated *service***impl.cxx** files where it contains the text: **TODO implement operation**.

Write a service operation implementation

You can write implementation code for a service operation by using boilerplate source code files. Write an implementation after you create a service operation and generate the service artifacts.

The *Services Reference* documents the C++ service APIs. This reference is available on Support Center.

1. **Create the operation** for which you need to write the implementation.

   In the **Extensions\Code Generation\Services** folder, open a service library. Right-click the service to hold the new operation, choose **Open**, click the **Operations** tab, and click the **Add** button.

2. **Generate the service artifacts**.

   Right-click the **Services** folder and choose **Generate Code→Service Artifacts**. To generate code only for a particular service library, select only the service library.

3. Locate the generated code files.

   To see the generated files, open the **Project Files\src\server**_service-library_ folder.

4. Switch to the **C/C++** perspective by choosing **Window**→**Open Perspective**→**Other**→**C/C++**.

5. Write the implementation.

   Write an implementation in the generated _service_**Impl.cxx** files.

   Right-click the **src\server**_service-library_****_service_**impl.cxx** file and choose **Open** or **Open With**. Add your implementation where it contains the following text:

   ```
   // Your implementation
   ```

   Replace **// Your implementation** with your implementation code, for example:

   ```
   /**
    * Saves the input bom windows. This method can be used to save
    * product structures that are created/modified using bom lines.
    *
    * @param    bomWindows  Array of bomwindows that need to be saved
    * @return   SaveBOMWindowsResponse retruns the ServiceData containing updated
    *                                  BOMWindow and list of partial errors.
    *
    */
   Cad::_2008_06::StructureManagementImpl::SaveBOMWindowsResponse
       Cad::_2008_06::StructureManagementImpl::saveBOMWindows (
           const vector< BusinessObjectRef< Teamcenter::BOMWindow > > &bomWindows )
   {
       SaveBOMWindowsResponse resp;
       ResultStatus rStat;
       // loop over the input set, processing each one
       for( vector< BusinessObjectRef< BOMWindow > >::const_iterator iter
            = bomWindows.begin();
            iter != bomWindows.end(); ++ iter )
       {
           const BusinessObjectRef< BOMWindow > &bomWindow = *iter;
           try
           {
               // validate the input
               if(bomWindow.tag() == NULLTAG)
               {
                   ERROR_store( ERROR_severity_error, error-constant );
                   resp.serviceData.addErrorStack ();
                   continue;
               }
               // call tcserver function to process this
               rStat = BOM_save_window( bomWindow.tag() );
               // include modified objects
               resp.serviceData.addUpdatedObject( bomWindow.tag());
           }
           // process errors as partial failures, linked to this input object
   ```

```
        catch( IFail & )
        {
              ERROR_store( ERROR_severity_error, error-constant);
              resp.serviceData.addErrorStack( bomWindow );
        }
    }
    return resp;
}
```

In the sample, *error-constant* is replaced with a constant that maps to the correct error message in the text server. Typically, you create your own error constants. Constants can be exposed through ITK (for server code customization) by addition of the header file in the **kit_include.xml** file.

> Caution:
>
> This sample is for example only and is not expected to compile in your customization environment. Do not copy and paste this sample code into your own code.

6. If you want the implementation files to be regenerated after making changes to the services, you must manually remove or rename the files and then right-click the service and choose **Generate Code→Service Artifacts**. The Business Modeler IDE does not regenerate the files if they exist, to avoid overwriting the implementation files after you edit them.

7. After you write implementations, **build server and client libraries**.

ServiceData implementation

The **ServiceData** class is the primary means for returning Teamcenter data model objects. Objects are sorted in *created*, *deleted*, *updated* or *plain* lists. These lists give the client application access to the primary data returned by the service. The Teamcenter Services server-side framework provides the service implementor access to the following **ServiceData** class with methods to add objects to the different lists (**tag_t** can be passed to these in place of the **BusinessObject** tag):

```
class ServiceData
{
public:
    void addCreatedObject ( const BusinessObjectRef<Teamcenter::BusinessObject> obj );
    void addCreatedObjects( const std::vector<
BusinessObjectRef<Teamcenter::BusinessObject>
       >& objs );
    void addDeletedObject ( const std::string& uid );
    void addDeletedObjects( const std::vector<std::string>& uids );
    void addUpdatedObject ( const BusinessObjectRef<Teamcenter::BusinessObject>
       obj );
    void addUpdatedObjects( const std::vector<
BusinessObjectRef<Teamcenter::BusinessObject>
       >& objs );
    void addUpdatedObject( const BusinessObjectRef<Teamcenter::BusinessObject> obj,
       const std::set<std::string>& propNames );
    void addPlainObject   ( const BusinessObjectRef<Teamcenter::BusinessObject> obj );
    void addPlainObjects  ( const std::vector<
BusinessObjectRef<Teamcenter::BusinessObject>
       >& objs );
```

```
        ...
    };
    …
        void addObject     ( const BusinessObjectRef<Teamcenter::BusinessObject> obj );
        void addObjects    ( const std::vector< BusinessObjectRef<Teamcenter::BusinessObject>
            >& objs );
    …
```

> **Note:**
>
> If the service is returning a plain object in a response structure, you must also add the object to the **ServiceData** class using the **addPlainObject** tag. Otherwise, only a UID is sent and the client **ModelManager** process can't instantiate a real object (which can cause casting issues when unpacking the service response). The framework typically automatically adds created, updated, and deleted objects to the **ServiceData**.

The nature of the Teamcenter data model is such that simply returning the primary requested objects is not enough. In many instances you must also return the objects referenced by primary objects. The **ServiceData** class allows this to be done with the **addObject** method. Objects added to the **ServiceData** class through the **addObject** method are not directly accessible by the client application; they are only accessible through the appropriate properties on the primary objects of the **ServiceData** class:

```
class ServiceData
{
public:
    ...
    void    addObject        ( const tag_t& obj );
    void    addObjects       ( const std::vector<tag_t>& objs );
};
```

The different add object methods on the **ServiceData** class only add references to the objects to the data structure. These object references are added without any of the associated properties. There are two ways for adding object properties to the **ServiceData** class: explicitly added by the service implementation, or automatically through the object property policy. To explicitly add properties use the **addProperty** or **addProperties** method:

```
class ServiceData
{
public:
    ...
    void addProperty  ( const BusinessObjectRef<Teamcenter::BusinessObject> obj,
        const std::string& propertyName    );
    void addProperties( const BusinessObjectRef<Teamcenter::BusinessObject> obj,
        const std::set<std::string>& propNames );
};
```

Partial errors implementation

The **ServiceData** class extends from the **PartialErrors** class to pick up methods to add the current error stack as a partial error. The **ERROR_store_ask** and **ERROR_store_clear** utilities are used to get the current stack and clear it. The partial error can optionally be added with a reference to a **BusinessObject** tag, a client ID, or index:

```
class ServiceData extends PartialErrors
{
public:
    void addErrorStack  (   );
    void addErrorStack  ( const std::string& clientId );
    void addErrorStack  ( const BusinessObjectRef<Teamcenter::BusinessObject> obj );
    void addErrorStackWithIndex ( int clientIndex );

};
```

Build server and client libraries

Building libraries is the final step in the service creation process. Before you build libraries, you must create service libraries, services, and service operations, as well as build service artifacts and write service implementation code. The Business Modeler IDE build system generates make files for building all the service artifacts.

1. Ensure that the code generation environment is set up in the project the way you want it.

    a.   Right-click the project and choose **Properties**.

    b.   Choose **Teamcenter→Build Configuration** and **Teamcenter→Code Generation**. You can change the default compiler options for each platform on the **Build Configuration** dialog box.

2. Select the project in the **Advanced** perspective and choose **Project→Build All** on the menu bar.

    You can also perform this step in the **C/C++** perspective. The **C/C++** perspective is an environment you can use to work with C and C++ files, as well as to perform build operations.

    The output libraries are generated for all the client bindings, and the server side library is built. Output library files are placed under the **output** folder in the **client**, **server**, and **types** subfolders.

3. After all services are built, you can package them into a template.

    On the menu bar, choose **BMIDE→Generate Software Package**.

    > Note:
    >
    > The Business Modeler IDE packages the built C++ library as part of its template packaging. The complete solution that includes the data model and the C++ run-time libraries are packaged together.

Teamcenter Services build output

After building all service artifacts, the following JAR files or libraries are created. Based on need, client applications should reference the corresponding client binding library or JAR file during integration and customization.

> Note:
>
> You define the bindings that are generated when you create a new project.

- Server libraries

  A server-side library is created with the name **lib***prefix-service-library-name.library-extension* in the **output\server\lib** folder.

  For example, if a prefix is set to **xyz3** and the service library name is **MyQuery**, the library file name is **libxyz3myquery.dll** for Windows servers and **libxyz3myquery.so** for Linux servers.

  This library must be deployed into the Teamcenter server using Teamcenter Environment Manager (TEM).

- Client language bindings

  Based on the options specified, the corresponding language client bindings are created for services. For Java alone there are two types of bindings: strong and loose. Only one of them should be used in a client application for calling services. For example, if a client application is a Java application and if you choose to use strong bindings, only the *prefix-service-library-name***Strong.jar** file needs to be in the classpath and not the *prefix-service-library-name***Loose.jar** file.

  C++ language client bindings have the **lib***prefix-service-library-name***strong.***library-extension* file name convention. For C++, two versions of strong libraries are created, one with smart pointers and one without (legacy). The smart pointers C++ library name has a **mngd** file name suffix. Only one of these two libraries needs to be used.

  C# language client bindings have a *prefix-service-library-name***strong.dll** file name convention.

- Client data model

  For any custom data model created in a custom template (that is, if there are any custom business objects created and used in service definitions), the Business Modeler IDE generates client-side data model (CDM) libraries for the selected languages and must be referenced in the client application.

  The file has the *prefix-model-solution-name.extension* naming convention.

  For example, if a prefix is set to **xyz3** and the solution name is **MyCustom**, the strong data model library file for C++ is named **libxyz3modelmycustom.dll** for Windows servers and **libxyz3modelmycustom.so** for Linux servers. For a Java client, a **xyz3StrongModelMyCustom.jar** file is created and for C#, **xyz3StrongModelMyCustom.dll** is created.

- Rich client bindings

A JAR file is created that can be used in rich client customization. The corresponding type library also must be placed into the classpath for consuming services in the rich client.

The JAR file name is *prefix-service-library-name***Rac.jar**.

- WSDL bindings

  WSDL files are also generated depending on the option specified, and the corresponding Axis bindings are created with a *prefix-service-library-name***Wsdl.jar** naming convention. This must be deployed onto the Web tier. After it is deployed, client applications can consume the services using WSDL definitions.

Following is typical usage (for example, if the prefix is **xyz3**, the solution name is **MyCustom**, and the service library name is **MyQuery**).

Deployment	Language / Technology	Libraries to use (Windows)	Libraries to use (Linux)
Server	C++	**libxyz3myquery.dll**	**libxyz3myquery.so**
Client	C++	**libxyz3myquerystrong.dll** **libxyz3modelmycustom.dll**	**libxyz3myquerystrong.so** **libxyz3modelmycustom.so**
Client	C++ - Smart pointer version	**libxyz3myquerystrongmngd.dll** **libxyz3modelmycustommngd.dll**	**libxyz3myquerystrongmngd. so** **libxyz3modelmycustommngd .so**
Client	Java - Strong	**xyz3MyQueryStrong.jar** **xyz3StrongModelMyCustom.jar**	**xyz3MyQueryStrong.jar** **xyz3StrongModelMyCustom.j ar**
Client	Java - Loose	**xyz3MyQueryLoose.jar**	**xyz3MyQueryLoose.jar**
Client	C#	**xyz3MyQueryStrong.dll** **xyz3StrongModelMyCustom.dll**	
Client	Rich client	**xyz3MyQueryRac.jar** **xyz3StrongModelMyCustom.jar**	**xyz3MyQueryRac.jar** **xyz3StrongModelMyCustom.j ar**

Deploy-ment	Language / Technology	Libraries to use (Windows)	Libraries to use (Linux)
Client	WSDL	Generate client bindings using exposed WSDL files.	Generate client bindings using exposed WSDL files.
Web tier	Java EE	**xyz3mycustomWsdl.jar**	**xyz3mycustomWsdl.jar**
Web tier	.NET	Use the TEM feature file.	

4. ITK customization

Introduction to the ITK

The Integration Toolkit (ITK) is a set of software tools provided by Siemens Digital Industries Software that you can use to integrate third-party or user-developed applications with Teamcenter. The ITK is a set of C functions used directly by Teamcenter and NX.

Reference information about ITK APIs can be found in the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is available on Support Center.)

> Note:
>
> Siemens Digital Industries Software recommends that instead of using ITK for server customization, you should use the data model customization method. The data model framework is the structure used by Teamcenter itself.

Basic concepts for ITK customization

Syntax conventions used in this guide

This guide 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
```

Format

All ITK functions have a standard format that attempts to give the most information possible in a small space. A template is shown below, followed by a more detailed description. All prototypes are located in include files named *classname***.h** for the class of objects that the functions operate on.

Additional information about specific functions can be found in the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is available on Support Center.)

The design intent for the format of ITK functions is to provide the maximum amount of information in a small space. The standard format is:

```
int module_verb_class_modifier ( const type variable-name[dimension] /* [I/O/OF]
*/ );
```

int	Nearly all ITK functions return an integer error code. This code can be passed to the **EMH_ask_error_text** function.
module	This is the module designator. Related classes are grouped together in modules. For example, the **Dataset**, **DatasetType**, **Tool**, and **RevisionAnchor** classes are all handled by the **AE** module. Other examples of modules are **PSM**, **POM**, **FL**, and **MAIL**.
verb	This is the first key word describing an action to be taken on an object or set of objects. Common actions are **create**, **add**, **remove**, **copy**, **set**, and **ask**.
class	This is the class name or an abbreviation of the class name. The exceptions are for modules that only deal with one class, like **Workspace** or modules that deal with many classes, like **WSOM** and **POM**.
modifier	This is a word or several words that give more description of how the action of the verb applies to the class. For example, in the **RLM_update_af_status** function, **status** indicates what is being updated in the **af** (authorization folder).
const	Input pointer variables which are not meant to be modified normally are declared with a **const** to ensure that they are not accidentally modified.
type	This is the data type of the argument (for example, **int**, **tag_t***, or **char*****).

variable-name	This is a variable name that is descriptive enough so a programmer does not need to look it up in the documentation.
dimension	This value is normally specified for arrays where the calling program is responsible for allocating space. They are normally specified with a constant definition of the form *module_description_*c. These constants should be used in dimensioning arrays or looping through the values of an array to make your programs more maintainable. However, you may see the values of these constants in the include files. This is useful so you do not establish naming conventions that leads to name strings longer than can be passed to the functions.
I/O/OF	These characters indicate whether the particular argument is input, output or output-free. Output-free means that the function allocates space for the returned data and this space should be freed with the **MEM_free** function.

Variable naming conventions

Variables in the interface are normally as descriptive as possible, consisting of key words separated by underscores (_).

- Typedefs end in **_t**.

- Constants end in **_c**.

- Enums end in **_e**.

Most typedefs and constants begin with the module designator like the function names to make it clear where they are meant to be used.

- **ITEM_id_size_c**

- **ITEM_name_size_c**

- **WSO_name_size_c**

> Caution:
>
> These fixed-size buffer ITK APIs are deprecated.

Many of the Integration Toolkit (ITK) APIs in Teamcenter are based on fixed-size string lengths. As long as Teamcenter is operating in a deployment scenario where each character occupies only one byte (such as western European languages), there are not any issues. However, when Teamcenter is deployed in a scenario where each character can occupy multiple bytes (such as Japanese, Korean, or Chinese), data truncation can occur. Therefore, the fixed-size buffer ITK APIs are deprecated, and wrapper APIs are provided for each of the fixed-size buffer APIs.

Siemens Digital Industries Software recommends that you refactor any customizations that use the older fixed-size buffer APIs and replace them with the new APIs. The new APIs use the **char*** type instead of fixed **char** array size in the input and output parameters. The new APIs use dynamic buffering to replace the fixed-size buffering method.

For a list of the deprecated ITK APIs and their replacements, see the **Deprecated** tab in the *Integration Toolkit Function Reference*.

Class hierarchy

Because Teamcenter is an object-oriented system, it is not necessary to have a function for every action on every class. Usually, functions associated with a particular class work with instances for any subclasses. For example, the **FL_** functions for folders work with authorizations and envelopes. The **AOM** module and the **WSOM** module consist of functions that correspond to the methods of the **POM_application_object** and **WorkspaceObject** abstract classes respectively. **AOM** functions work for all application objects and **WSOM** functions work for all workspace objects.

> Caution:
>
> When you create objects, an implicit lock is placed on the newly created object. After you perform an **AOM_save**, it is important that you call **AOM_unlock**.

The same mechanism described here also enables standard Teamcenter functions to work on subclasses that you may define. For example, if you define a subclass of the **folder** class, you can pass instances of it to **FL** functions.

Include files

Required header file

All ITK programs *must* include either **tcinit/tcinit.h** or **tc/tc_startup.h**. These contain the definitions of many standard datatypes, constants, and functions that almost every ITK program requires, such as **tag_t** and **ITK_ok**.

Use **tcinit/tcinit.h** if your code calls:

- ITK_init_from_cpp
- ITK__initialize_tc
- ITK_init_module
- ITK_auto_login

Use **tc/tc_startup.h** for everything else.

Other headers

The include files are located in subdirectories of the *TC_ROOT*/**include** directory. You must include the subdirectory when calling the include file. For example, when calling the **epm.h** include file, which is located in the **epm** subdirectory, you must call it as:

```
#include <epm/epm.h>
```

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

For example, you could use this statement:

```
#include <sa/sa.h>
```

Or you could use these statements:

```
#include <sa/am.h>
#include <sa/sa_errors.h>
#include <sa/person.h>
#include <sa/role.h>
#include <sa/group.h>
#include <sa/groupmember.h>
#include <sa/user.h>
#include <sa/site.h>
```

There is also an error include file for every module called *module*_**errors.h** or sometimes *classname*_**errors.h**. These files all contain error offsets from a value assigned in the **tc/emh_const.h** file. One exception is POM, which has its errors in the **pom/pom/pom_errors.h** file.

All of the other ITK functions have their prototypes and useful constants and types in a file called *classname*.**h**. In addition there are header files for many of the major modules that contain constants and include all of the header files for the classes in that module. Sometimes this header file is required, sometimes it is just convenient. The *Integration Toolkit Function Reference* tells you which header files are required.

Teamcenter standard classes

Teamcenter also comes with a set of include files that mimic standard C include files.

For example, instead of using:

```
#include "stdarg.h"
```

You could use:

```
#include <fclasses/tc_stdarg.h>
```

These include files are used in Teamcenter to insulate it from the operating system. Sometimes the files include the system files directly, but other times system dependencies are handled in the **tc_** include files, thus enabling the Teamcenter source code and yours to remain system independent. Some of the files provided are:

fclasses/tc_ctype.h
fclasses/tc_errno.h
fclasses/tc_limits.h
fclasses/tc_math.h
fclasses/tc_stat.h
fclasses/tc_stdarg.h
fclasses/tc_stddef.h
fclasses/tc_stdio.h
fclasses/tc_stdlib.h
fclasses/tc_string.h
fclasses/tc_time.h
fclasses/tc_types.h

Special data types

All objects in the ITK are identified by tags of C type **tag_t**. A run-time unique identifier isolates the client code using the object from its representation in memory, making it safe from direct pointer access.

Tags may be compared by use of the C language **==** and **!=** operators. An unset **tag_t** variable is assigned the **NULLTAG** null value. If two tags have the same value, they reference the same object. For example:

```
#include <tc/tc.h>
#include <ae/ae.h>
{
   tag_t dataset_tag;
   int error_code;
   login, etc. ...
   error_code = AE_find_dataset ("my special dataset", &dataset_tag)
   if( error_code ==ITK_ok )
      { if (dataset_tag == NULLTAG)
         { /*
            * Could not find my special dataset, so go create one.
            */
            error_code = AE_create_dataset_with_id (....
         }
      }
   else
      {
         report error ....
```

Basic tasks for ITK customization

Configure Visual Studio to build and debug ITK applications

Overview

The Teamcenter ITK build tools are provided in the form of script files such as **compile.bat** and **linkitk.bat**. These script files do not take advantage of the features available in Visual Studio IDE, such as debugging, code completion, symbol browsing, and project management. You use these files to configure Visual Studio to build debug and release versions of an ITK application and to configure the debugger to debug the application as follows:

Prerequisites:

- A Teamcenter installation

- The Teamcenter sample files installed using Teamcenter Environment Manager (TEM)

- The Microsoft Visual Studio IDE

> Note:
>
> Consult the Hardware and Software Certifications knowledge base article on Support Center for the required version levels of third-party software.

Create the Visual Studio solution

The various files you need are located in the Teamcenter sample files directory, *TC_ROOT***sample**.

1. Create a Visual Studio solution for your ITK code using one of the provided sample files.

   For example: **...\utilities\find_recently_saved_item_rev_itk_main.c**

2. Set up the project property sheets to compile and link the ITK sample code.

   Use the **compile** and **link** script files to provide the necessary directories, dependencies, preprocessor definitions, and libraries to your solution.

You may now build the solution. The resultant executable file must be run from within a Teamcenter command-line environment.

Configure the debugger

The Microsoft Visual Studio IDE provides a debugger for your C++ code. Because the debugger runs the code within Visual Studio, the proper Teamcenter environment is not available. Before you start the

debugger, the Teamcenter environment variables must be set for your application to run correctly, **TC_ROOT** and **FMS_HOME** for example.

1.   Copy the environment variables from an existing Teamcenter command-line environment.

2.   Paste them into the **Environment** field on the debug property page for the debug configuration.

You can also add **Command Arguments** to save typing. For example:

```
-u=user -p=password
```

Debugging ITK

Journal files

Teamcenter optionally journals the calls to all of the ITK functions. This is in a file called **journal_file** after the program is run. This file can be very useful in finding out what your program has done and where it went wrong. There are many journaling flags in the system for the modules and ITK in general. Also, there are flags for **NONE** and **ALL**. Another option is to set the **TC_JOURNAL** environment variable to **FULL**, **SUMMARY**, or **NONE**.

For POM, use the **POM_set_env_info** function; otherwise, use the *xxx*_**set_journaling(int** *flag***)** function, where *flag* is equal to 0 (zero) for off or not equal to 0 for on. You can have ITK and EPM journaling on if you are working in that area and leave POM and PS journaling off, ensuring that your journal file does not get cluttered with unuseful information. The following code shows how the POM logs input and output arguments:

```
POM_start ( 'aali',  'aali',  'Information Manager', user,  topmost_class_id,
pom_version)
@*        --> 6s
@*        FM_init_module
@*           --> 8s
@*            EIM_PM_id_of_process ( 103,  process_id)
@*           process_id = 558 returns 0
@*        returns 0 [in 2 secs]
@*        AM_init_module
@*           FM_init_module
@*           returns 0
@*            FM_allocate_file_id ( file)
@*           returns 0, file = 000127cc08b2
@*           --> 9s
@*            AM_identify_version ( identity)
@*           returns 0
@*        returns 0
@*        EIM_PM_id_of_process ( 105,   process_id)
@*        process_id = 559 returns 0
@*        --> 10s
@*        FM_read_file ( 00650002)
@*        returns 0 @*        --> 11s
@*        FM_ask_path ( 00650002,  750,   path)
@*        returns 0, path = '/home/demov2/user_data/d1/f000627cbd90f_pom_schema'
@*        --> 12s
```

```
@*         FM_unload_file ( 00650002)
@*         returns 0
@*          --> 14s
@*         FM_allocate_file_id ( file)
@*         returns 0, file = 000227cc08b2
@* user = 006500e0 <QAscMZ60AAAAyD>, topmost_class_id = 006500d0, pom_version = 100
returns 0 (in 16 secs)
```

The following code shows another example of a journal file:

```
POM_name_of_class ( 00650021,  class_name)
@* class_name = 'Signoff' returns 0
POM_class_id_of_class ( 'Signoff',  class_id)
@* class_id = 00650021 returns 0
POM_describe_class ( 00650021, NULL,  application_name,  descriptor,  n_attrs,  attr_ids)
@* application_name = 'Information Manager', descriptor = 0, n_attrs = 4,
@*     attr_ids = { 00650022,  00650023,  00650024,   00650025 } returns 0
POM_describe_attrs ( 00650021,  4, /
{ 00650022,  00650023,  00650024,  00650025 },  names,  types,   max_string_lengths,
referenced_classes,  lengths,  descriptors,   attr_failures)
@*
@*     names = { 'comments',  'decision',  'group_member',   'decision_date' },
@*     types = { 2008,  2005,  2009,  2002 },
@*     max_string_lengths = { 240,  0,  0,  0 },
@*  referenced_classes = { 00000000 <AAAAAAAAAAAAAA>, 00000000 <AAAAAAAAAAAAAA>,
00650054, 00000000 <AAAAAAAAAAAAAA> },
@*     lengths = { 1,  1,  1,  1 },
@*     descriptors = { 64,  0,  0,  64 },
@*     attr_failures = { 0,  0,  0,  0 } returns 0
POM_free ( 0041ac28)
@* returns 0
POM_subclasses_of_class ( 00650021,  n_ids,  subclass_ids)
@* n_ids = 0, subclass_ids =  {}
@* NULL array pointer, length 0
@*  returns 0
```

The first few hundred lines of a journal file contain a lot of text like the example above. This is the result of reading in the schema. If you define **POM** classes, you can see this information in there also. This is helpful in determining if you have in the database what you think you have in it. Also, when you see class IDs or attribute IDs elsewhere in the journal file, you can refer to this section to see what they are.

The following code is an example of an application function being journaled:

```
    RLM_ask_approver_roles ( 006500e1)
    @*         POM_length_of_attr ( 006500e1,   00650009,  length)
    @*         length = 1 returns 0
    @*         POM_length_of_attr ( 006500e1,   00650009,  length)
    @*         length = 1 returns 0
    @*         POM_ask_attr_tags ( 006500e1,   00650009,  0,  1,  values,  is_it_null,
    is_it_empty)
    @*   values = { 006500dc <AAlccPghAAAAyD> }, is_it_null = { FALSE },
    @*         POM_free ( 0041cf50)
    @*         returns 0
    @*         POM_free ( 0041d390)
    @*         returns 0
    @* role_count = 1, approver_roles = { 006500dc <AAlccPghAAAAyD> } returns 0
```

The journal information for lower level routines are nested inside the higher level routines.

System logs

The system log can be useful in diagnosing errors because when a function returns an error, it often writes extra information to the system log. For example, if your program crashes you may not have an error message, but the end of the system log may show what the program was trying to do just before crashing.

Logging

Information on logging can be found in the Log Manager (LM) module reference section of the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Error handling

Teamcenter stores errors on a central error stack. An error may be posted at a low level and each layer of the code handles the error reported by the level below, potentially adding a new error of its own to the top of the stack to add more contextual information to the exception being reported. This stack of errors is what you see displayed in the Teamcenter error window in the user interface. ITK functions always return the top error from the stack if there is one. If the call is successful, **ITK_ok** is returned.

The **Error Message Handler** (**EMH**) ITK functions enable you to access the full error stack and decode individual Teamcenter error codes into the internationalized texts that are defined in the XML code and displayed in the error windows at the user interface. **EMH** ITK functions are defined in the **tc/emh.h** header file. They give the ability to store errors, access errors on the stack, and decode error codes into internationalized text.

Additional information on **EMH** functions can be found in the **EMH** section of the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is available on Support Center.)

> Note:
>
> It is good practice to acknowledge that you have received and handled an error reported by an ITK call by clearing the error store with a call to the **EMH_clear_errors** function. However, if you do not do this it normally does not cause a problem, as all initial errors should start a new stack with a call to the **EMH_store_initial_error** function.

> Caution:
>
> ITK extension rules cannot provide information or warning feedback to users.

For example, if you want a warning dialog box to appear after an operation is done, you cannot define the warning in ITK extension rule code because of the code processing order:

1. Teamcenter code is executed.

2. The custom code is executed.

3. Teamcenter code is executed.

For example, you put the following into the extension rule custom code:

```
EMH_store_error_s2(EMH_severity_warning, ITK_err,
            "EMH_severity_warning", "ITK_ok");
        return ITK_ok;
```

You place this code in step 2 because you want the Teamcenter framework to proceed with step 3 so that when the whole execution is done, the user receives a warning message in the client.

However, the only errors or warnings returned from a service operation are the ones explicitly returned by the business logic of the service implementation. This is typically done with code like this:

```
        Try
        {
                ....
        }
        Catch( IFail& ifail)
        {
                serviceData.addErrorStack();   // This will add
the
current IFail and whatever other errors/warnings are currently on
the
error store
        }
```

The Teamcenter Services Framework does not look at the error store to see if there are warnings or errors. Unless the exception moves up to the Teamcenter (service implementation) code where it is caught, the error or warning is lost.

Memory management

Frequently, memory is allocated by a Teamcenter function and returned to the caller in a pointer variable. This is indicated in the code by the **/* <OF> */** following the argument definition. This memory should be freed by passing the address to the **MEM_free** function. For example, to use the function:

```
int AE_tool_extent (
int* tool_count, /* <O> */
tag_t** tool_tags /* <OF> */
);
```

You should include the following code:

```
{
    tag_t* tool_tags;
    int tool_count;
    int error_code;
    int i;
    error_code = AE_tool_extent &tool_count, &tool_tags);
    if( error_code == ITK_ok )
{
    for (i = 0; i < tool_count; i++)
    {...}
    MEM_free (tool_tags);
}
else
{
        report the error
}
...
```

Initializing modules

You must either call the initialization function for all modules, **ITK_init_module**, or the individual module initialization function, such as **FL_init_module**, before using any of the functions in that module. If you do not, you get an error like **FL_module_not_initialized**. The only reason that a module initialization should ever fail is if the module requires a license and there is no license available.

You should also remember to exit modules. In some cases this could cause significant memory to be freed. It may also free up a concurrent license for use by another user.

Compiling

Compile your program using the ANSI option. The ITK functions can also be called from C++. Sample compile scripts are available in the *TC_ROOT*/**sample** directory. Assuming the script name is **compile**, it can be used by executing the following command.

TC_ROOT/**sample**/**compile** *filename*.c

The actual compile statements are as follows:

- **HP**

```
cc filename.c -c -Aa filename.o -D_HPUX_SOURCE -DHPP -I$TC_INCLUDE
```

- **Solaris**

  ```
  cc -KPIC -DSUN -DUNX -D_SOLARIS_SOURCE -DSYSV +w    -Xc -O -I. -I$
  {TC_INCLUDE} -c filename.c -o filename.o
  ```

- **Linux**

  ```
  gcc -c -fPIC -m64 -DPOSIX -I${TC_INCLUDE} filename.c -o filename.o
  ```

- **Windows**

  If you are using Windows, you must supply the **-DIPLIB=none** argument to tell the compile script that a stand-alone program object is being compiled:

  ```
  $TC_ROOT/sample/compile -DIPLIB=none filename.c
  ```

> **Note:**
>
> For information about system hardware and software requirements for Teamcenter, see the Siemens Digital Industries Software Certification Database:
>
> **http://support.industrysoftware.automation.siemens.com /certification/teamcenter.shtml**

Linking user exits and server exits

Link user exits for Linux

User exits are places in the server where you can add additional behavior by attaching an extension.

Additional information about ITK user exits can be found in the *Integration Toolkit Function Reference*.

> **Note:**
>
> The *Integration Toolkit Function Reference* is available on Support Center.

To verify your Linux environment and installation:

1. Set the Teamcenter environment. Set the **TC_ROOT**, **TC_LIBRARY**, and **TC_INCLUDE** environment variables.

2. Create a **user_exits** and code directories.

3. Extract the objects with the following commands:

```
$ cd user_exits
$ ar -x $TC_ROOT/lib/libuser_exits.a
```

4. Complete your customization. It is good practice to include a **printf** statement with a coding version and compile date. The following is a minimal example:

```
$ cd ../code
```

   a. Copy all of the files that you want to modify (**user_gsshell.c** in the following example):

```
$ cp $TC_ROOT/sample/examples/user_gsshell.c .
```

   b. Make the required changes.

   For example, include your own **printf** statement in the **USER_gs_shell_init_module** function in the **user_gsshell.c** file:

```
printf("\nCustom library for %s, version %s, compiled on %s %s\n
\n",
customer, version, _DATE_, _TIME_);
```

   c. Create additional **.c** files as required, but see below for Windows considerations when exporting new symbols from the library.

5. Compile your library objects and move them into the library directory:

```
$ $TC_ROOT/sample/compile *.c
$ cp *.o ../user_exits
```

6. Move into the library directory and link the library:

```
$ cd ../user_exits
$ $TC_ROOT/sample/link_user_exits
```

7. You now have a new **libuser_exits.s** file. Set the **TC_USER_LIB** command to get Teamcenter to use it:

```
$ export TC_USER_LIB='pwd'
$ $TC_BIN/tc
```

   You see your **printf** messages after the Teamcenter copyright messages.

If the **libuser_exits.so** file compiles without undefined symbols, then the installation and environment are correct.

Link user exits in Windows

User exits are places in the server where you can add additional behavior by attaching an extension.

Additional information about ITK user exits can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

To verify your Windows environment and installation:

1. Set the Teamcenter environment. Set the **TC_ROOT**, **TC_LIBRARY**, and **TC_INCLUDE** environment variables.

2. Create a **user_exits** and code directories.

3. Extract the objects with the following commands:

   ```
   $ cd user_exits
   $ extract_objects %TC_ROOT%\lib\libuser_exits.ar.lib
   ```

4. Complete your customization. It is considered good practice to include a **printf** statement with a coding version and compile date. The following is a minimal example:

   ```
   $ cd ../code
   ```

   a. Copy all of the files that you want to modify (the **user_gsshell.c** file in the following example):

   ```
   $ cp $TC_ROOT/sample/examples/user_gsshell.c .
   ```

   b. Make the required changes.

   For example, include your own **printf** statement in the **USER_gs_shell_init_module** function in the **user_gsshell.c** file:

   ```
   printf("\nCustom library for %s, version %s, compiled on %s %s\n
   \n",
   customer, version, _DATE_, _TIME_);
   ```

   c. Create additional **.c** files as required, but see below for considerations when exporting new symbols from the library.

5. Compile your library objects and move them into the library directory:

```
%TC_ROOT%\sample\compile -DIPLIB=libuser_exits *.c
copy *.obj ..\user_exits
```

6. Move into the library directory and link the library. You do not need **.def** files:

```
cd ../user_exits
 %TC_ROOT%\sample\link_user_exits
```

7. You now have new **libuser_exits.dll** and **libuser_exits.lib** files. Set the **TC_USER_LIB** command to get Teamcenter to use it:

```
for /f "delims==" %i in ('chdir') do set TC_USER_LIB=%i
%TC_BIN%\tc
```

> Note:
>
> To run stand-alone programs against this **libuser_exits.dll** library, you must link them against the corresponding **libuser_exits.lib** file.

You see your **printf** messages after the Teamcenter copyright messages.

Export new symbols from libuser_exits.dll on Windows

Windows requires all symbols exported from **.dll** files to be declared. This mechanism is invisible to Linux, which exports every extern as usual (so you do not need two sets of source).

In each header file, declare new functions for export from the **.dll** file:

1. Enter **#include <user_exits/libuser_exits_exports.h>** as the last **#include** of the header file before all declarations.

2. Enter **USER_EXITS_API** after the word **extern** to mark each function declaration for export. This is called decoration.

3. Enter **#include <user_exits/libuser_exits_undef.h>** at the end of the header file after all declarations (within any multiple include prevention preprocessing).

In each source file defining new functions for export from the **.dll** file, ensure that the **.c** file includes its own corresponding **.h** file, so the definitions in the **.c** file know about the decorated declarations in the **.h** file. If you do not do this, the compiler complains about missing symbols beginning with **__int__** when you try to link stand-alone code against the **libuser_exits.lib** file.

Now you can compile and link as normal (in other words, compile with **-DIPLIB=libuser_exits**).

Linking stand-alone programs

When you link your program, you must include the **itk_main.o** file on the command line. This file contains the main object for your ITK program. To make this work, all of your ITK programs must begin like this:

```
int ITK_user_main(int argc, char* argv[])
   { int result;
     /*
      * We should use ITK_auto_login here.
      */
     if( (result = ITK_auto_login()) == ITK_ok)
        {
             do what you need to do ...
             result = ITK_exit_module();
        }
     return result;
   }
```

The order of Teamcenter libraries is important or your code may not link. Also add any additional object files you have created to the link script to include them in the main executable. A sample **linkitk** script file exists in the *TC_ROOT*/**sample** directory.

If the script name is **linkitk**, you can execute the script with the following command:

```
$TC_ROOT/sample/linkitk –o executable-name filename.o
```

> Note:
>
> - The **linkitk.bat** file contains only the core Teamcenter libraries. To call server APIs from optional solutions, you must add the libraries from the optional solutions to the **linkitk.bat** file. You must also ensure these libraries are available in the **TC_LIBRARY** location.
>
>   For example, to add the libraries for the optional Content Management solution, add the following lines near the end of the **linkitk.bat** file (before the **"-out:$EXECUTABLE.exe"** line):
>
>   ```
>   "$ENV{TC_LIBRARY}\\libcontmgmt.lib /delayload:libcontmgmt.dll ".
>   "$ENV{TC_LIBRARY}\\libcontmgmtbase.lib /
>   delayload:libcontmgmtbase.dll ".
>   "$ENV{TC_LIBRARY}\\libcontmgmtbasedispatch.lib
>   /delayload:libcontmgmtbasedispatch.dll ".
>   ```
>
> - For Linux, add the *TC_ROOT***lib** folder to the **LD_LIBRARY_PATH** system environment variable to link custom programs, for example:
>
>   ```
>   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ TC_ROOT/lib
>   ```

Run the executable

To run or execute the program, you must have Teamcenter system privileges (in other words, an account on Teamcenter). If your program is called without the **ITK_auto_login** function, supply the arguments for user, password and group. If the **ITK_auto_login** function is used, the necessary arguments can come from the command line. The specific style for entry is dependent on your code. In this class, the standard procedure is as follows:

```
filename –u=user –p=password –g=group
```

Be careful to execute this command in a directory where additional files can be created since session and log files are created at run time.

Upgrading your ITK programs

If you are upgrading from an earlier version of Teamcenter to the current version of Teamcenter, you may need to change your ITK programs. All obsolete ITK functions need to be replaced in your programs. For the list of obsolete functions, see the Teamcenter 14.2 **README** file.

If you are using any deprecated functions, you should replace those functions as soon as you can, because deprecated functions will be removed from a future version. A list of deprecated functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Batch ITK

Batch program template

The following code shows a template for writing a batch program:

```
#include <tc/tc.h>
#define WRONG_USAGE 100001
void dousage()
{
   printf("\nUSAGE: template -u=user -p=password -g=\"\" \n");
   return;
}
int ITK_user_main(int argc, char* argv[])
{
   int status;
   char* usr = ITK_ask_cli_argument("-u=");/* gets user */
   char* upw = ITK_ask_cli_argument("-p=");/* gets the password */
   char* ugp = ITK_ask_cli_argument("-g=");/* what the group is */
```

```
   char* help = ITK_ask_cli_argument("-h=");/* help option  */
   char *s;
   if ( (help) )
   {
      dousage();
      return WRONG_USAGE ;
   }
   /* Example of must have... */
   if (( !usr) || (!upw) || (!ugp))
   {
      dousage();
      return WRONG_USAGE ;
   }
   ITK_initialize_text_services (0);
   status = ITK_init_module( usr,upw,ugp);
   if ( status != ITK_ok)
   {
     EMH_ask_error_text( status, &s);
     printf("Error with ITK_init_module: %s \n",s);
     MEM_free(s);
     return status;
   }
   else printf("Logon to 'template' succesful as %s.\n", usr);
   JOURNAL_comment ("Preparing to list tool formats");
   TC_write_syslog("Preparing to list tool formats \n");
   /* Call your functions between here */
   ITK_exit_module( TRUE);
   return status;
}
```

Compile and link batch programs

Follow this sequence to generate executables for batch programs.

> Caution:
>
> Make sure the **TC_INCLUDE** and **TC_LIBRARY** environment variables are set correctly.

1.   Compile the batch program with the **compile** script:

```
compile –DIPLIB=none filename.c
```

The **-DIPLIB** argument specifies the target library; use **none** for batch programs.

2.   Link your program with the system library with the **linkitk** script:

```
linkitk –o executable filename.obj file3.obj file2.obj
```

3.   Execute the batch program:

   • If you use **ITK_auto_login**:

       filename arguments

   • If you use **ITK_init_module**:

       filename –u=*user* –p=*password* –g=*group* *arguments*

The **compile** and **linkitk** scripts are Teamcenter-specific scripts and are available in the *TC_ROOT*/**sample** directory.

Custom exits

Custom exits environment

Custom exits (also known as supplier custom hooks) allow nonconflicting customizations to co-exist.

> Caution:
>
> You should use the Business Modeler IDE user exits to do this kind of customization work. Only create your own custom exits if you cannot accomplish your goals with user exits in the Business Modeler IDE.

All the user exit customizations of the site are built in the form of a single library, called *site-name*.**dll/so/sl**.

For example, if the site name is **cust1**, the library is **cust1.dll/so/sl**. The custom library *site-name*.**dll/so/sl** is linked with **libuser_exits**. Use the **link_custom_exits.bat** file from the *TC_ROOT*/**sample** directory. All custom stand-alone ITK utilities are linked with the *site-name*.**dll/so/sl** file instead of **libuser_exits** as the customization is in the *site-name*.**dll/so/sl** file. To see the customization, define the custom library name in the **TC_customization_libraries** preference. Make sure the prefix and the library prefix are the same. Set the preference in the **Options** dialog box, accessed from the **Edit** menu in My Teamcenter.

Build the library file

1.   Create a working directory (for example, **cust1**).

2.   Copy all your user exits and server exits customization code to this directory.

3.   Compile your object files inside this directory using the *TC_ROOT*/**sample/compile** command with the **-DIPLIB=NONE** option.

4. Build the dynamic linked/shared library with the **link_custom_exits** command. For example:

```
link_custom_exits cust1
```

> Note:
>
> - Do not modify or rebuild the **libuser_exits** and **libserver_exits** libraries with the custom code.
>
> - Do not unarchive the **libuser_exits.ar.lib** file. These object files are not required to be linked with the **cust1** library.
>
> - Any custom library that uses the **USERARG_*** ITK method requires dependency over the COTS (standard) **libict** shared library. If you build this custom library on the Linux platform, you must modify the $*TC_ROOT*/**sample/link_custom_exits** script to register dependency of your custom library on the **libict** library. For example, add the following lines to the **link_custom_exits** script:
>
> ```
> set optionalLibraries="-lict"
>     $LinkTC -o $customLibrary $objectFiles $optionalLibraries
> ```
>
> This ensures that the custom library is loaded correctly even by those standalone ITK programs that do not have explicit dependency over the **libict** library.

5. If you are using Windows, copy the **.dll** and **.PDB** files to the directory that contains the **.dll** files. For example:

```
copy cust1.dll %TC_ROOT%\\bin\\.
copy cust1.pdb %TC_ROOT%\\bin\\.
```

If you are using Linux, copy the shared library to the **TC_USER_LIB** directory or put the **cust1** directory in the shared library path.

Register custom callbacks

Create a *site-name*__register_calls.c_ file in the site's source code development area. The following code shows an example:

```
#include <tccore/custom.h>
#include <user_exits/user_exits.h>
#include your-include-files
extern DLLAPI int cust1_register_callbacks()
{
 CUSTOM_register_exit ("site-name", "Base User Exit Function",
(CUSTOM_EXIT_ftn_t)
your-custom-function-name);
}
```

- *site-name*__register_callbacks__

    This is the entry point to the custom library. This function is executed after loading the custom library. Ensure that the *site-name* name is the same as the value stored in the **TC_customization_libraries** preference. For example, if the custom library is **cust1**, this function should be called **cust1_register_callbacks**.

- *site-name* argument to the **CUSTOM_register_exit**

    site-name is also the same as the value stored in the **TC_customization_libraries** preference.

- Base user exit function

    This string is the name of the base user exit against which the custom function is registered. All the **USER_** functions from the files provided in the *TC_ROOT*/**sample** directory can be customized.

- *your-custom-function-name*

    This is the function pointer registered against the above base user exit. The definition of this function does not need to exist in this file, it could be in another file.

- *your-include-files*

    Add any custom include files needed during the compilation and linking of the custom library.

Register user exits customization

Custom hooks provide you the ability to:

- Execute only site customization

- Execute site customization along with other customizations

- Execute the default core functionality

The custom functions registered in the *site-name*__register_calls.c__ file should have an integer pointer as the first argument. This integer pointer should return the decision, indicating whether to execute only the current customization, execute all customizations, or only the default core functionality. The following variables are defined in the **custom.h** file in the *TC_ROOT*/**include/tccore** directory:

```
#define ALL_CUSTOMIZATIONS 2
#define ONLY_CURRENT _CUSTOMIZATION 1
#define NO_CUSTOMIZATIONS 0
```

The following code shows an example with the following conditions:

- The site name is **cust1**.

- The custom library is **cust1**.

- A custom function is registered against the **USER_new_dataset_name**.

```
cust1_register_calls.c:
#include <tccore/custom.h>
#include <user_exits/user_exits.h>
#include <cust1_dataset.h>
#include <cust1_register_callbacks.h>
#include <cust1_register_callbacks.h>
extern DLLAPI int cust1_register_callbacks()
{
CUSTOM_register_exit ("cust1", "USER_new_dataset_name", (CUSTOM_EXIT_ftn_t)
CUST1_new_dataset_name);
}
cust1_register_callbacks.h:
extern DLLAPI int cust1_new_dataset_name (int *decision, va_list args);
cust1_dataset.c:
#include <cust1_register_callbacks.h>
#include <tccore/custom.h>
extern DLLAPI int CUST1_new_dataset_name (int *decision, va_list args)
{
  /*
  Expand the va_list using va_arg
  The va_list args contains all the variables from USER_new_dataset_name.
  */
  tag_t  owner = va_arg (args, tag_t);
  tag_t  dataset_type = va_arg (args, tag_t);
  tag_t  relation_type = va_arg (args, tag_t);
  const char* basis_name = va_arg (args, const char *);
  char**  dataset_name = va_arg (args, char **);
  logical*  modifiable = va_arg (args, logical *);
  /*
   if cust1 wants all other customizations to be executed, then set
   *decision = ALL_CUSTOMIZATIONS;
  elseif cust1 wants only its customization to be executed then set
  *decision = ONLY_CURRENT_CUSTOMIZATION;
  else if cust1 wants only base functionality to be executed, then set
   *decision = NO_CUSTOMIZATIONS;
  */
  /* Write your custom code
 */
  return ifail;
}
```

> Note:
>
> - The **va_list** argument in the **CUSTOM_EXIT_ftn_t** function should match the arguments of the base user exit or server exit function in the **CUSTOM_register_exit** function.
>
> - To register pre- or post-actions against Teamcenter messages, you must use the **USER_init_module** method, for example:
>
>   ```
>   CUSTOM_register_exit("lib_myco_exits", "USER_init_module",
>   (CUSTOM_EXIT_ftn_t)register_init_module);
>   ```
>
>   For a pre- or post-action to be invoked during a call to a Teamcenter Services operation, the custom extension must be registered against the **USER_init_module**.

Execute multiple customizations

If your site wants to execute some other site's customization other than your customization, follow these steps:

1. Both the **cust1** and **cust2** sites should follow the sections above to register their customization for **USER_** exits.

2. Both sites should build their **cust1.dll/sl/so cust2.dll/sl/so** custom libraries to share them.

3. Neither site should rebuild the **libuser_exits** and **libserver_exits**.

4. Add the custom library names in the **TC_customization_libraries** preference in the **Options** dialog box, accessed from the **Edit** menu in My Teamcenter:

    ```
    TC_customization_libraries=
    cust1
    cust2
    ```

5. The preference values should match the library names. For example, the **cust1** site should have the **cust1.dll/sl/so** file and the **cust2** site should have the **cust2.dll/sl/so** file.

6. The custom libraries of the **cust1** and **cust2** sites (which would be **cust1.dll/sl/so** and **cust2.dll/sl/so**, respectively) should be in the library path:

    - **LD_LIBRARY_PATH** for Solaris or Linux

    - **PATH** for Windows

Register custom exits

CUSTOM_register_callbacks

This function registers customizations for all customization contexts registered in the **Options** dialog box (accessed from the **Edit** menu in My Teamcenter):

```
int CUSTOM_register_callbacks ( void )
```

This function loads the custom library and calls the entry point function pointer to register custom exits. The entry point function contains the custom registrations for the required **USER_** exits.

CUSTOM_register_exit

This ITK function should be called only in the **USER_preint_module** function in the **user_init.c** file. It should not be called anywhere else. This function registers a custom exit (a custom function pointer) for a given **USER_** exit function:

```
int CUSTOM_register_exit ( const char*  context,    /* <I> */
const char*  base_ftn_name,  /* <I> */
CUSTOM_EXIT_ftn_t custom_ftn  /* <I> */
);
```

- *context*

  Specifies the context in which this custom exit has to be registered. It is the name of the customization library (for example, GM, Ford, Suzuki).

- *base_ftn_name*

  Specifies the name of the **USER_** exit for which the custom exit must be registered (for example, **USER_new_dataset_name**).

- *custom_ftn*

  Specifies the name of the custom exit (a custom function pointer).

CUSTOM_execute_callbacks

This function executes the custom callbacks registered for a particular **USER_** exit:

```
int CUSTOM_execute_callbacks ( int* decision, /* <O> */
const char* ftn_name, /* <I> */
variables        /* <I> */
);
```

- *decision*

  Executes one of the following options:

  - **ALL_CUSTOMIZATIONS**

  - **ONLY_CURRENT_CUSTOMIZATION**

  - **NO_CUSTOMIZATIONS**

- *ftn_name*

  Name of the **USER_** exit.

- *variables*

  The variables that need to be passed to the custom exit (a custom function pointer).

For each custom library registered in the **Options** dialog box, accessed from the **Edit** menu in My Teamcenter:

- Get the corresponding custom function pointer.

- Execute the custom function with the arguments passed.

This function is called in all the **USER_** functions in user exits and server exits. The **va_list** list must be expanded in the custom function.

CUSTOM_execute_callbacks_from_library

This function executes custom callbacks registered in a particular library for a particular **USER_** exit by:

- Getting the corresponding custom function pointer.

- Executing the custom function with the arguments passed.

  ```
  int CUSTOM_execute_callbacks_from_library ( int* decision, /* <O> */
  const char* lib_name, /* <I> */
  const char* ftn_name, /* <I> */
  variables          /* <I> */
  );
  ```

- *decision*

  Executes one of the following options:

- **ALL_CUSTOMIZATIONS**

- **ONLY_CURRENT_CUSTOMIZATION**

- **NO_CUSTOMIZATIONS**

- *lib_name*

  Specifies the name of the customization context.

- *ftn_name*

  Specifies the name of the **USER_** exit.

- *variables*

  Specifies the variables that need to be passed to the custom exit (a custom function pointer).

The **va_list** list must be explained in the custom function.

Example

The following code (using **cust1** as the site) shows registering a custom handler in a custom exit:

```
extern int cust1_action_handler(EPM_action_message_t msg) {
  /* insert custom handler code here */
}
extern DLLAPI int cust1_register_custom_handlers(int *decision, va_list
args) {
    int rcode = ITK_ok;
    *decision  = ALL_CUSTOMIZATIONS;
    rcode = EPM_register_action_handler("cust1-action-handler",
"",
     cust1_action_handler);
    if (rcode == ITK_ok)
        fprintf(stdout, "cust1-action-handler successfully registered!\n");
    else
        fprintf(stdout, "WARNING - cust1-action-handler NOT registered!
\n");
    return rcode;
}
extern DLLAPI int cust1_register_callbacks () {
   CUSTOM_register_exit ( "cust1", "USER_gs_shell_init_module",
       (CUSTOM_EXIT_ftn_t)  cust1_register_custom_handlers);
   return ( ITK_ok );
}
```

Register run-time properties using custom hooks

There is a limitation on registering property methods on the same type. Among the registered property methods on the same type, the top one in the preferences stored in the database overrides the registration of the one that follows. Therefore only one registration works.

To work around this, register multiple methods (one as base method and the rest as **post_action** methods) on the same type by following these steps:

1. Register one property method as base method, and the rest as **post_action** methods.

2. To register the property methods, follow this sample registration in the following code:

```
USER_prop_init_entry_t icsTypesMethods[] =
      {
         { (char *)"WorkspaceObject", icsInitWorkspaceObjectProperties, NULL
}
      };
    int npTypes = sizeof ( icsTypesMethods )/
sizeof( USER_prop_init_entry_t );
    for ( i = 0; i < npTypes; i++ )
    {
      // firstly, findout if a base method is already registered for
      // TCTYPE_init_user_props_msg
      ifail = METHOD__find_method( icsTypesMethods[i].type_name,
      TCTYPE_init_user_props_msg, &initUserPropsMethod );
      if( ifail == ITK_ok)
      {
        if ( initUserPropsMethod.id != 0 )
        {
           // already registered, the add the method as a post_action method
          // add method as post action to the existing base method
          ifail = METHOD_add_action( initUserPropsMethod,
                      METHOD_post_action_type,
                      icsTypesMethods[i].user_method,
                      icsTypesMethods[i].user_args );
        }
        else
        {
           // not yet registerd, add the method as a base method
          // there is no method registerd
          ifail = METHOD_register_method ( icsTypesMethods[i].type_name,
                        TCTYPE_init_user_props_msg,
                        icsTypesMethods[i].user_method,
                        icsTypesMethods[i].user_args,
                        &methodld );
        }
```

```
        }
    }
```

Core ITK functions

File relocation

Introduction to file relocation

When you create a dataset in Teamcenter, a file is created in the user's volume directory. Once it is released (in other words, signoff is performed), it remains in the same volume along with other non-released objects. To list and backup all of the files related to the released datasets for all of the users, all of the dataset files should be copied to a separate directory.

Two programs have been developed to perform this task. One activates an action handler to relocate the files on release of the job. The other relocates all of the existing released jobs.

Relocating datasets by registering an action handler

To relocate a dataset after it is released, you need to register an EPM action handler with the **EPM_register_action_handler** function. Input parameters should be action strings and function callbacks.

A handler is a small program used to accomplish a specific task. There are two kinds of handlers:

• Rule

• Action

> Note:
>
> This function should be called in the **user_gsshell.c** file.

To add an action handler:

1.   Run Teamcenter and log on as a system administrator.

2.   Open **Workflow Designer**.

3.   Choose **File→New Root Template**.

     The **New Root Template** dialog box is displayed.

4.   Give the new root template a name and select another root template to base this template on. Click **OK**.

5. Click the button that displays the task handlers pane.

6. Select the task action where you want the handler to execute, and then select the handler.

7. Select the arguments needed for your handler and type their values.

8. Click the plus button **+** to add the handler to the task action. When you are done, close the **Handlers** dialog box.

Sample programs for relocating datasets

The following sample programs relocate files associated with datasets being released. It assumes that the **FLRELDIR** directory exists on the operating system. If any of the target objects are datasets, all files referenced by it are copied to the operating system directory specified by **FLRELDIR**.

It is assumed, for these examples, that the **usr/tmp/release** directory exists on the system and that all users have write permission to that directory.

> Note:
>
> These functions relocate only the files associated with the dataset. If the job is released for an item, folder, form and dataset, then this function only relocates the files for the job released for the dataset.

- **smp_cr_mv_hl.c**

  Contains the **USER_cr_init_module** function. This function should be called in the **USER_gs_shell_init_module** function.

  Once called, a new **SMP-auto-relocate-file** action handler is registered and the job is released. If it contains a dataset, the dataset is relocated to the **FLRELDIR** directory.

- **smp_cr_mv_fl.c**

  Contains the **relocate_released_dataset_file** function. This function's purpose is to relocate files associated with the datasets being released. Using the tag of the release job as input, it finds all target objects. If any of the target objects is a dataset, all files referenced by it are copied to the operating system directory specified by **FLRELDIR**. The directory is not created by this function.

- **sample_cr_move_files_main.c**

  Works as a standalone ITK program. It logs into Teamcenter and initializes the AE, PSM and SA modules. Next, it creates the **FLRELDIR** operating system directory. It then searches for all released jobs containing datasets and relocates these dataset files to the **FLRELDIR** directory.

Following are compilation tips:

- Compile and execute **smp_cr_mv_hl.c** and **smp_cr_mv_fl.c**

  To compile these files, add the **USER_cr_init_module** function to the **USER_gs_shell_init_module** function of the **user_gsshell.c** file (supplied by Teamcenter). Compile the **smp_cr_mv_hl.c**, **smp_cr_mv_fl.c** and **user_gsshell.c** files separately. Create a new **libuser_exits.sl** file using the above **.o** files.

  To execute this file, add the **SMP-auto-relocate-file** action handler to all of the release procedures. Once the job is released, it copies released datasets to the **FLRELDIR** operating system directory.

- Compile and execute **sample_cr_move_files_main.c** and **smp_cr_mv_fl.c**

  Compile these files separately. Link the two files using the **linkitk** command and create the **sample_cr_move_files_main** executable. You can execute this file from the command line using:

  ```
  sample_cr_move_files_main –u=user-id –p=password –g=group
  ```

  The *user-id* variable is the Teamcenter user name; *password* is the Teamcenter user password; and *group* is the Teamcenter user group name.

Change the revisioning scheme

If you want to change the default item revisioning scheme to something else (for example, revisions **A**, **B**, **C**, and so forth to revisions **-a**, **-b**, **-c**, and so on), you can modify the sample code in the **user_part_no.c** file.

Performing this type of customization is only required if you have overly complex revision scheme that cannot be done in the Business Modeler IDE using naming rules or revision naming rules.

General hints about property coding

Memory allocation

If you need to manually allocate memory when writing code for the Teamcenter server, use **MEM_alloc** instead of **malloc**.

MEM_alloc is part of the **BASE_UTILS_API** group, one of several functions that parallel standard C memory allocation functions. Use **MEM_free** to release memory back to the system.

More information about these and other functions can be found in the Integration Toolkit Function Reference, which is part of the References for Administrators and Customizers.

Method arguments

Method arguments extracted using **va_arg** must be extracted in the sequence indicated in the *Integration Toolkit Function Reference*. For example:

```
tag_t prop_tag = va_args(args, tag_t);
char **value = va_args(args, char**);
```

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

ITK property messages

The following messages are used to set property values:

- **PROP_init_value_msg**

  Initializes a property with a specified value.

  > Caution:
  >
  > A method registered against this message does not exist for most objects. Therefore the **METHOD_find_method**, **METHOD_add_action**, and some other functions cannot be used unless another method is registered. The **METHOD_find_method** function does not consider this an error and does not return a failure code. It merely returns a null **method_id**.

- **PROP_ask_value_*type*_msg**

  Returns the value of a property.

  > Note:
  >
  > This message is called by most ITK functions and internally by Teamcenter. However, some ITK functions (for example, Form, FL, AE, and so on) may be bypassing this code at this time.

  > Caution:
  >
  > Never call the **PROP_ask_value_*type*** function from inside a method registered against the **PROP_ask_value_*type*_msg** function on the same property. This forces Teamcenter into a recursive loop. Instead use the **PROP_get_value_*type*** function to get the raw data value.

- **PROP_set_value_*type*_msg**

  Is passed whenever the user chooses **OK** or **Apply** in the **Attributes** dialog box or sets a value in the workspace, Structure Manager, or with ITK. The **PROP_assign_value_*type*** function can be used.

> Caution:
>
> Never call the **PROP_set_value_**_type_ function from inside a method registered against the **PROP_set_value_**_type_**_msg** function on the same property. This forces Teamcenter into a recursive loop. Instead use the **PROP_assign_value_**_type_ function to store the raw data value.

Value caching

Value caching is a technique used with run-time properties to improve performance. This is valuable when the value of a derived property is not likely to change during the current Teamcenter session. To cache values, use the following technique:

1. The first time a run-time property is called using the **PROP_ask_value_string** method, cache the string value with the **PROP_assign_**_type_ method.

2. On subsequent **PROP_ask_value_string** method calls, use the cached value by calling the **PROP_get_**_type_ method.

Create a custom bulk loader

Overview

Bulk property retrieval loads all the requested properties for a list of objects in one request, significantly reducing the number of database trips. Bulk property retrieval is used by the Teamcenter services framework to improve property retrieval performance for all service operations.

A run-time property computes its value when requested based upon other persistent properties. It may run a database query to retrieve various persistent objects and their properties from the database. A run-time property can register a bulk loader function with the bulk property retrieval framework. Bulk property retrieval calls the bulk loader functionality with all of the objects upon which the run-time property is requested, so that the bulk loader can perform bulk query and bulk loading of persistent objects.

To implement a custom bulk loader:

1. Register the bulk loader.

2. Implement the bulk loader.

3. Implement the **BulkPropertyContextData** subclass to cache the bulk query results.

4. Modify the getter function to access the bulk loader cache.

Register the bulk-loader

Using the Business Modeler IDE, define a **PostAction** extension, and then attach it to the legacy operation: **IMANTYPE_init_intl_props** on the **BOMLine** business object. Please refer to Attaching extensions to operations for details.

In the extension function, call the following C API to register the custom bulk loader for the custom runtime property.

```
int PROPDESC_register_bulk_loader( tag_t pdTag,
                                   logical invokeOnce,
                                   BulkProperty_bulk_loader_t bulkLoaderFn )
```

The following example is a code snippet.

```
Int registerBulkLoaderPostAction(METHOD_message_t *, va_list args)
{
   int ifail = ITK_ok;
   va_list largs;
   va_copy( largs, args );
   tag_t type_tag = va_arg( largs,tag_t);
   va_end( largs );
   TCTYPE_ask_name( typeTag, typeName );
   if( strcmp( typeName, "BOMLine" ) == 0 )
   {
      //Register the same bulk loader for all the properties of each absocc Form BO
      TCTYPE_ask_property_by_name(type_tag, propNames[i], &pd );
      logical invokeOnce = true;
      PROPDESC_register_bulk_loader( pd,
                                     invokeOnce,
                                     pipeAbsoccBulkLoaderFn );
   }
}
```

Implement the bulk loader

The bulk loader function signature is defined as follows:

```
/**
The bulk-loader function type (signature)
*/
typedef void (*BulkProperty_bulk_loader_t)(
   tag_t pdTag, /**< (I) tag of PropertyDescriptor */
   int n_tags, /**< (I) number of object tags to retrieve the property from*/
   const tag_t* tags /**< (I) array of object tags */
   Teamcenter::Property::BulkPropertyContextData** userData /**<O> For caching results
(optional) */
);
```

The following example is a code snippet.

```
void pipeAbsoccBulkLoaderFn(tag_t pdTag, int n_tags, const tag_t* tags,
```

```
Teamcenter::Property::BulkPropertyContextData** bulkContextData)
{
    //Build a bulk query to find Pipe Absocc form tags for all the input BOMLines
    //Call query API

    //Bulk-load Absocc Forms
    ifail = AOM_load_instances( n_absOccForms, absOccForms, &n_loaded, &loaded);

    //Implement PipeBulkContextData to maintain a std::map of bomline tag and absocc tag
    *bulkContextData = new PipeBulkContextData();
    (*bulkContextData)->set( n_tags, bomLineTags, absOccForms );
}
```

Implement the BulkPropertyContextData subclass to cache the bulk query results

A bulk loader needs to cache the bulk query results and enable the getter function to access the results. In addition, the bulk query results should be cleared at the end of the bulk property retrieval to avoid any stale data. For these reasons, the cache data should be returned to the bulk property retrieval framework to enable the cache data to be cleared by the system at the end of the bulk property retrieval. This requires the custom cache to be implemented as a subclass of **BulkPropertyContextData**.

```
Class AbsoccBulkContextData : public Teamcenter::Property::BulkPropertyContextData
{
public:
    AbsoccBulkContextData ();
    virtual ~AbsoccBulkContextData(); //clear m_bomLineAbsoccTagMap;
    void set( n_tags, tag_t* keyTags, tag_t* valueTags );
    tag_t find( tag_t key );
private:
    std::map m_bomLineAbsoccTagMap;
}
```

Modify the getter function to access the bulk loader cache

The getter function can use the following C APIs to access the bulk loader results if they exist.

```
logical PROPDESC_is_bulk_loading_context()
BulkPropertyContextData* PROPDESC_ask_bulk_prop_context_data( METHOD_message_t* m)
```

The following example is a code snippet.

```
Int getPipeAbsoccPropFn(METHOD_message_t* m, va_list args)
{
    if( PROPDESC_is_bulk_loading_context() )
    {
        AbsoccBulkContextData*
                    pipeBulkContextData = static_cast<AbsoccBulkContextData *>
                                        (PROPDESC_ask_bulk_prop_context_data( m ));
        if( pipeBulkContextData != 0 )
        {
            absoccFormTag = pipeBulkContextData->find( bomLineTag );
        }
    }
    else
    {
```

```
        //Existing query code to find the absocc form
    }
    //Check if abosoccFormTag can be NULLTAG
}
```

Persistent Object Manager layer

Persistent Object Manager

Persistent Object Manager concepts

The Persistent Object Manager (POM) forms part of the data management layer of the ITK. POM provides the interface between Teamcenter objects and the relational database management system (RDBMS). POM provides:

- **Data manipulation services**

  Manipulation of in-memory objects and support for their saving and retrieval to and from the underlying RDBMS.

- **Locking**

  Support for many different applications accessing the same data concurrently.

- **Referential integrity**

  Protection against the deletion of data used by more than one application.

- **Access controls**

  Support for the access control lists attributed to objects. The access controls themselves are manipulated by functions provided by the Access Manager.

To understand how the Persistent Object Manager (POM) works, you need to know about classes, attributes, instances, indexes, and tags:

- Attribute
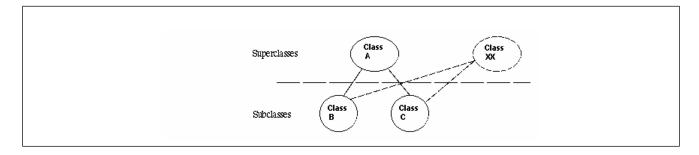
  Attributes are the individual data fields within classes. Each attribute has a unique name within the class that identifies it. It also has a definition that defines the type of data object and its permitted values. For example, a Human has attributes of weight and eye color. Weight is a real number. Eye color may be one of a set (for example, blue, grey, brown).

  You can declare attributes to be class variables. This is an attribute that has the same value for all instances of the class. When the attribute value is changed by modifying any instances, the value changes for all instances of the same class (or subclass). This is also known as a static variable.

• Class

A class is a definition of a data structure containing one or more attributes. Classes are organized in hierarchy, with superclasses and subclasses. A class inherits attributes from its parent (superclass) and passes its attributes on to its children. POM allows only single inheritance; that is, any class can have only one parent, one grandparent, and so on.

In the following figure, classes B and C can inherit from (in other words, be direct subclasses of) only one immediate superclass, class A. Classes B and C cannot also inherit from class XX.

Classes

Each class has a unique name. For example, the **Human** class is a subclass of **Primate**, which is itself a subclass of **Mammal**. **Primate** inherits attributes such as hairiness and warmbloodedness from the definition of **Mammal**. In addition to the inherited attributes, it adds its own, such as binocular vision. As **Human** inherits from **Primate**, it therefore has all the attributes of a **Primate**. In addition, it can have specific attributes, such as I.Q.

• Index

Indexes are created on one or more of the attributes that exist within a class definition. Indexes can only be defined across attributes that belong to the same class.

Indexes aid performance. For example, two attributes that the **Human** class has are weight and eye color. To list all Humans who have an eye color of brown, you would probably search the database for all instances of the **Human** class that matches the applied criteria.

Though this type of search is performed well by a RDBMS, it is slow. The alternative is to have an index on the eye color attribute. Internally, the database maintains its index of where to find objects with the required attributes. Therefore, the search for Humans with brown eyes is much faster if an index is put on a value attribute.

To optimize performance when defining the indexes, you have to anticipate what searches the application is likely to be required.

> Note:
>
> Indexes improves query performance when searching on that indexed attribute, but at a small time penalty for save, update, and delete operations which must update the instance and the index.

An index can be defined as unique. This ensures that the POM checks for an existing value for an index when a class with attributes containing an index is instantiated. The uniqueness is tested only upon a save to the database. While an instance remains in a user's session, duplicates may exist.

For example, the **POM_user** class has a name attribute that has a unique index defined on it. This prevents two POM users with the same name from being registered with the POM.

When the unique index spans more than one attribute, the uniqueness must be for all the attributes combined. For example, the **POM_user** class with **first_name** and **second_name** attributes, both covered with a unique index, allows (John Smith and John Brown) and (John Smith and Bill Smith), but not (John Smith and John Smith), to be registered with POM.

- Instance

  Instances are the individual data items created from class definitions. Making an instance from a class definition is instantiating that class. Each instance has the attributes defined in the class, but has its own values for these attributes. For example, an individual Human might have a weight value of 160 pounds and an eye color value of brown.

  Some classes are defined to be uninstantiable, meaning they cannot be instantiated. An example of such a class would be Mammal.

- Tag

  A tag is a short-lived reference to an instance. The tag persists for the length of a session and can be exchanged between processes in a session.

  To exchange a reference to an instance between processes of different sessions, the tag must be converted to a persistent identifier called a handle by the sender, and from handle to tag by the receiver. The receiver's tag can be different in value to the sender's as they are in different sessions.

POM data manipulation services

The data manipulation services allow you to execute various actions on instances.

> Note:
>
> Do not use these methods on **WorkspaceObject** business objects. Only use these methods if a more specific function is not available.

- **POM_class_of_instance**

  Finds the class to which an instance belongs. This function returns the class in which the instance was instantiated.

  Use the **POM_loaded_class_of_instance** function to find the class of an instance as it has been loaded.

- **POM_copy_instances**

  Creates copies of a set of existing instances that must be loaded in the current session. The new instances are of the same class as the source. The attributes are maintained as follows:

  - Application-maintained attribute values are copied.

  - System-maintained attribute values (for example, **creation_date**) are set appropriately.

  The copy is created as an instance of the class in which the original instance was instantiated. This applies even if the original instance is loaded as an instance of a superclass.

- **POM_create_instance**

  Creates an instance of a specified class.

  > Caution:
  >
  > Only use this method when you cannot use the
  > **Teamcenter::BusinessObject::create(Teamcenter::CreateInput*)?** method.

  You can start the POM module without setting a current group. Because of this, it is also possible that an attempt can be made to create an instance for which an owning group cannot be set. When this happens, the function fails with no current group.

  A function fails with an improperly defined class when the class is only partially defined. For example, if you define an attribute to have a lower bound but have not yet set the value for this lower bound.

  When an instance is created, all attributes with an initial value defined are set to that value. System-maintained attributes, such as creation date and owning user, are set by the system. All other attributes are empty. These must be set to allowable values before the instance can be saved.

- **POM_delete_instances**

  Enables the deletion of a specified instance. The **POM_delete_instances_by_enq** function enables the execution of a specified enquiry followed by the deletion of the resulting instances from the database.

> Note:
>
> The instances to be deleted must not be referenced or loaded by this or any other POM session. This is how the POM maintains referential integrity. Newly created instances (in other words, those that have not yet been saved) cannot be deleted. This is because deletion is a database operation and these newly, unsaved instances do not yet exist there. Unloading these instances effectively deletes them. If a class is *application-protected*, then instances of that class can only be deleted by that application.
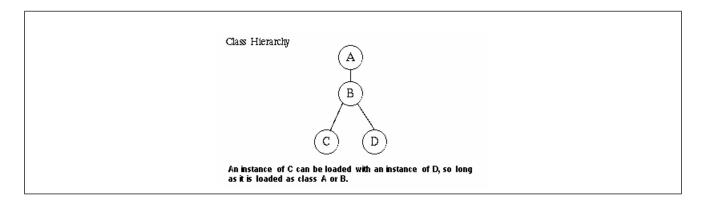
- **POM_instances_of_class**

  Returns all the instances of a specified class.

- **POM_load_instances**

  Loads and creates complete instances of a class. To load instances that are of different classes, use the **POM_load_instances_any_class** function. However, the class that they are loaded as must be common to all loaded instances as in the following figure.

  Instances can be loaded as instances of their own class.

Class hierarchy

When the **modify_lock** argument for this function is set, the instances are locked for modification, therefore no other session can lock them for further modification at the same time. When they are not locked for modification but are either set to **read_lock** or **no_lock**, this permits other sessions to lock them for modification or read as required.

The **POM_load_instances_by_enq** function executes a specified enquiry and loads the resulting set of instances from the database, creating in-memory copies of the instances.

The **POM_is_loaded** function checks whether a specified instance is loaded in the caller's local memory. Newly created instances are counted as loaded, as they are already present in the local memory having been created during that session.

To check whether the given instance is loaded in the caller's local memory for modification, use the **POM_modifiable** function.

- **POM_order_instances**

  Orders an array of loaded POM instances on the values of attributes that are common to all the instances.

  For this operation to be successful, the following criteria must be applied:

  - The instances must be loaded.

  - Only one attribute may be specified for ordering.

  - None of the attributes being ordered can be empty.

- **POM_refresh_instances**

  Refreshes a specified set of instances when instances are all of the same class.

  The refresh operation is the equivalent to unloading and then reloading an instance without having to complete the two separate operations. This operation causes the attribute values to be refreshed to the corresponding values in the database.

  Occasions when the refresh operation can be used are:

  - When you need to update the instance that is currently being worked on to reflect changes that may have occurred since the user's copy was made.

  - When you decide to scrap all changes that have been made to the instance.

  To refresh a specified set of instances that are not all of the same class, use the **POM_refresh_instances_any_class** function. The instances are refreshed in their actual class.

  These refresh functions allow the specified instance's locks to be changed.

  The **POM_refresh_required** function checks whether a specified instance, which is loaded for read in the caller's local memory, requires a refresh by testing if the database representation of the instance has been altered.

- **POM_save_instances**

  Saves to the database created instances or changes that were made to a set of instances. An additional option of this function is the ability to discard the in-memory representation.

The **POM_save_required** function checks whether a specified instance was modified. The instance may be newly created or loaded for modification in the caller's local memory storage. The function determines modification by testing whether the local memory representation of the instance was altered since the instance was loaded.

After an object instance is saved, it remains locked. The application must unlock the object by calling the **POM_refresh_instances** function.

- **POM_unload_instances**

  Unloads specified instances from the in-memory representation without saving any changes that were made to the in-memory representation.

  To save changes that were made to the in-memory representation, use the **POM_save_instances** function.

- **POM_references_of_instance**

  Enables you to find all instances and classes in the database which contain references to a specified instance.

  The level of the search through the database is determined by the **n_levels** argument. The **where_to_search** argument determines which domain to search (in other words, local memory, database or both).

  The **DS** argument limits the search to the working session (things not yet committed to the database). The **DB** argument limits the search to the database only. **DS** and **DB** together extends the search to both domains.

  When a class is found to be referencing an instance, there is a class variable which contains that reference.

POM inquiries to find instances

An inquiry to the Persistent Object Manager (POM) is a pattern that can be applied to saved instances of a specified class (and its subclasses) to select a subset of those instances.

The inquiry takes the form of restrictions on attribute values and combinations of the same, using the logical operators **AND** and **OR**. For example, if you have a furniture class with a color attribute and a table subclass with a height attribute, it is possible to create an inquiry to list all instances of the table class that has the restriction that the color is red.

Once an inquiry is created, it can be combined with other inquiries (such as height less than three feet). It can then either be executed (to return a list of tags of those instances that match the inquiry) or it can be passed to the **POM_load_instances_by_enq**, **POM_select_instances_by_enq**, and **POM_delete_instances_by_enq** functions which then load, select, or delete, as appropriate, those instances which satisfy the inquiry.

The following functions enable the user to create, but not to execute, an inquiry of a specified attribute or array-valued attributes:

- **POM_create_enquiry_on_**_type_

- **POM_create_enquiry_on_**_type_**s**

To execute a specified inquiry and return a list of tags of instances as specified by the inquiry, use the **POM_execute_enquiry** function.

To combine two inquiries use the **POM_combine_enquiries** function.

The **POM_order_enquiry** function specifies the order in which to produce the instances.

- Each attribute may be ordered in ascending or descending order. The instances are ordered on the value of each attribute in turn, starting with the first attribute in the array.

- This function can be used to override a previously specified order. Therefore, for instances, the same **enquiry_id** can be used more than once to produce different orderings of the same instances.

> Note:
>
> It is not possible to order on either variable length arrays (VLA) or arrays.

The **POM_negate_enquiry** function negates a specified enquiry and the **POM_delete_enquiries** function deletes specified enquiries.

POM attribute manipulation

Introduction to POM attribute manipulation

Attribute manipulation in the Persistent Object Manager (POM) enables you to modify attributes in the database, set and get attributes of loaded instances, manipulate tag and class IDs, manipulate variable length arrays, and check references.

> Note:
>
> Use these methods only if you must manage attributes outside the scope of a business object.

Modifying attributes in the database

The following group of functions enables you to change the attribute values of an instance without first loading the instance:

- **POM_modify_**_type_

- **POM_modify_**_type_**s**

- **POM_modify_null**

- **POM_modify_nulls**

- **POM_modify_**_type_**_by_enq**

- **POM_modify_**_type_**s_by_enq**

- **POM_modify_null_by_enq**

- **POM_modify_nulls_by_enq**

You can change the following attribute values:

- A specified attribute to a given value or to NULL for specified instances all in the same class.

- All or some of the specified array-valued attribute to the given array of values, or to NULL, for specified instances in the same class.

- The specified attribute to the given value for all instances found by an enquiry.

- All or some of the specified array-valued attributes to the given array of values for all instances found by an enquiry.

These functions modify data directly in the database and not on loaded instances.

Setting attributes of loaded instances

The following functions enable the user to set the value of a specified attribute in an instance:

- The **POM_set_attr_**_type_ and **POM_set_attr_null** functions change the specified attribute to the specified value or to NULL in the local memory only, for instances when the specified instances are all in the same class.

- The **POM_set_attr_**_type_**s** and **POM_set_attr_nulls** functions change all or some of the specified array valued attribute to the specified array of values or to NULL, in the local memory only, for instances all in the same class.

When changing the specified array-valued attribute to NULL, the attribute must exist for all the instances, therefore there must be a class with that attribute and instances must be in that class or in a subclass of it. This also applies to nonarray attributes.

Getting attributes of loaded instances

You can use the following functions within the POM module to retrieve the value of a specified attribute in an instance, for loaded instances only:

- **POM_ask_attr_***type*

  Returns the value of the attribute for a specified instance. The value returned is the value of that instance in the local memory; this could differ from the corresponding value in the database.

- **POM_ask_attr_***type***s**

  Returns n-values of values of the array-valued attribute for a specified instance starting from the position start. The values returned are those of the value of that instance in the local memory; these could differ from the corresponding values in the database.

Tag, class_id, and attr_id manipulation

The POM provides the following functions for conversion and comparison operations:

- **POM_tag_to_string**

- **POM_string_to_tag**

- **POM_compare_dates**

Variable length array manipulation

Variable length arrays (VLA) are attributes that can have a variable number of elements of their declared type. The data type of the VLA and class type (for typed references) or string length (for notes and strings) are declared when the attribute is defined.

When an instance of a class with a VLA attribute is created, that VLA attribute is initialized to a zero length.

The following functions are used to extend or remove elements from the VLA. All these functions, with the exception of **POM_reorder_attr** because it can be applied to any array, are specific to VLAs:

- **POM_insert_attr_{***type***}s**

  Inserts the specified values into a specified VLA attribute at a specified position. The maximum value for the specified position is the length of the VLA.

- **POM_clear_attr**

  Clears all values from the specified VLA, effectively setting its length to 0 (zero).

- **POM_remove_from_attr**

  Removes a specified number of elements from the VLA.

- **POM_append_attr_{*type*}s**

  Adds the specified values to the end of the specified VLAs.

- **POM_length_of_attr**

  Returns the length of a specified VLA (in other words, the number of values in the VLA).

- **POM_reorder_attr**

  Reorders elements within a VLA.

Check reference

To perform reference checking to find the type of a specified attribute of an instance, use the **POM_check_reference** function.

The **POM_check_reference** function permits checking for consistency between the class definition and the instance information.

POM users and groups

Introduction to POM users and groups

Persistent Object Manager (POM) user, group, and member objects allow users to be members of one or more groups. Users, groups, and members are maintained internally by instances of the **POM_user**, **POM_group**, and **POM_member** classes. A user is associated with a group by the existence of a member instance that links the user to the group. These instances must be saved to the database (with the **POM_save_instances** function) before they become usable.

A specified member of a group has the privilege of being the group administrator. This empowers that user to add or remove other members to or from that group. The system administrator or any member of the system administrator group has this privilege for any group.

Users and groups in the POM are objects in a protected application, the individual instances can only be manipulated by the authorized application (see the **POM_register_application** and **POM_identify_application** functions). System administrators that have been called to perform the changes can create new users and delete old users.

> Note:
>
> If you create a new member object for yourself, you cannot immediately set your current group (using the **POM_set_group** function) to that specified. You must save the member object first.

User, group and member objects are instances of application-protected classes, though some of the attributes of these instances are declared to be public read so that they can be asked about using the normal calls. The effect of making these classes application protected is to restrict access to certain attributes (for example, changing the password of a user is only possible by someone who knows the old password) or an system administrator and to prevent every user of the system from being able to generate new users or delete bona fide users.

Create

The following functions create instances of users, groups, and members:

- **POM_new_user**

- **POM_new_group**

- **POM_new_member**

The group administrator of the group in which the member resides, as well as the system administrator, can create members of a group.

Users must be associated with a group using the member functions before the user and group become registered.

Initialize

The following functions initialize users, groups, and members:

- **POM_init_user**

- **POM_init_group**

- **POM_init_member**

Only system administrators are allowed to use the these functions.

If a new instance of the **POM_user** class is created, using the **POM_new_user** function, the attributes of the instance are automatically initialized. However, if an system administrator creates a subclass of the **POM_user** class, the attributes (for example, names, password, and so on) of any new instance of the subclass must be initialized. Call the **POM_init_user** function to perform this initialization.

The same rules apply to creating groups and members.

Delete

The following functions delete users, groups and members:

- **POM_delete_user**

- **POM_delete_group**

- **POM_delete_member**

These functions are required because of the application protection. They take tags of instances of groups, users or members as appropriate or any subclass of the same. If the subclass is also application protected, then that application must also be identified.

Only a system administrator can use these functions.

> Note:
>
> When a user logs into the POM, the appropriate user and group objects are locked so that they cannot be deleted. This lock is updated whenever the user changes the group.

After a member object is deleted, the user specified cannot log onto the listed group.

Set group

To set the group to the named group, use the **POM_set_group** function. Alternatively, the **POM_set_group_name** function sets a specified name for a specified group.

The **POM_set_default_group** function sets the default group for the current user. The user object must be loaded for modification and then must be saved again afterwards for this change to be permanent.

The **POM_set_user_default_group** function sets the tag of the specified user's default group. This is the group which the user is logged into if the **POM_start** function is given an empty string for the **group_name** argument. Only a system administrator or that individual user can use this function.

To enable a user that is logging into a group to have group system administrator privileges, use the **POM_set_group_privilege** function with the privilege value of 1. A privilege value of 0 (zero) ensures that a person logging into the specified group only has the privileges associated with an ordinary user. The **POM_set_member_is_ga** function sets the group administration attribute for the specified group to the specified value, either 0 (zero) or 1.

A member is a loaded form of the database link that gives the user authority to log onto the chosen group (in other words, gives the user membership to the group).

The **POM_set_member_user** function sets the user (tag) attribute of the specified member instance to the supplied user tag. The **POM_set_member_group** function sets the group (tag) attribute of the specified member instance to the supplied group tag.

Set user status and license level

The **POM_set_user_status** function sets the status of the user. If the user is not active, set the **new_status** argument to **1**; if the user is active, set it to **0**. Only the system administrator can use this function. The status is not interpreted by the POM, it is provided for the system administrator to classify users.

The **POM_set_user_license_status** function sets the license level of the user. This function displays the number of licenses purchased and the number in use.

User and group inquiries

- The **POM_ask_owner** function returns the owning **user_tag** and **group_tag** for an instance.

- **POM_is_user_sa** advises if the current user is logged into the POM under a privileged group.

- **POM_ask_default_group** returns the tag for the default group of the current user.

- In contrast to the previous function, the **POM_ask_user_default_group** function returns the currently set default group for a user. Only a system administrator or the individual user can use this function. The user object must be loaded before using these functions.

- The **POM_ask_group** function returns the name of the group, as handed to the **POM_set_group** or **POM_start** functions, and the **group_tag**, as returned by the **POM_set_group** function.

- The **POM_get_user** function returns the name of the logged in user, as handed to the **POM_start** function, and the **user_tag**, as returned by the **POM_start** function.

Application protection

Classes can be associated with a particular application, so that only that application can perform the following operations on instances of these classes:

- Set

- Ask

- Delete

The following operations are not affected by application protection:

- **Create**

  The instance is of no use unless the attribute is completed, and it is not possible to do this without using set.

- **Load**

  Ask is required to get at the information.

- **Save**

  This fails if there are any empty attributes, and set is required to fill them in.

- **Modify**

  Modify is the equivalent to load, set, save, and unload.

The application protection affects all attributes defined in the given class in all instances of it or its subclasses. Attributes can be unprotected by declaring them as public, read, or modify in the attribute definition descriptor.

The association between a class and an application is not inherited by subclasses of the class. Those subclasses may have an association with some applications. This could be the same one as the class itself.

The **POM_identify_application** function is used by an application to identify itself to the POM so that the POM allows it to perform protected operations on classes with which the application is associated. On completion of the protected operations, the application becomes anonymous. POM does not allow further operations without the application re-identifying itself.

For an application to be registered with POM and thereby obtain the application ID and code that is required by the **POM_identify_application** function, use the **POM_register_application** function.

The code number returned by the **POM_register_application** function is a random number which, in turn, is used as a level of security checking.

POM system utilities

Starting and stopping POM

The **POM_start** function is the first call to the POM module and therefore it logs the user into the POM. If you call any other function before this function, then that function fails except for **POM_explain_last_error** which always works.

To log out of the POM, use the **POM_stop** function.

Password manipulation

To set the password for the specified user, use the **POM_set_password** function.

> **Note:**
>
> Because of the severity of this operation, it can only work on one user at a time.

The **POM_check_password** function checks the password for a user by comparing the given password with the stored password.

For security, the stored password is encrypted and never decrypted or returned through the interface. The encryption algorithm is not public.

> **Note:**
>
> When Security Services are installed and the **TC_SSO_SERVICE** environment variable is set in the **tc_profilevars** file, passwords are managed by an external identity service provider rather than Teamcenter. In this case, the **POM_set_password** function returns **POM_op_not_supported** (the ability to change a password in the rich client and ITK is disabled).

Rollback

The rollback facility enables you to restore a process to a previous state. The state at which a process is restored is dependent on the position of the specified markpoint.

Markpoints are used at your discretion and are placed using the **POM_place_markpoint** function. When a markpoint is placed, an argument is returned advising you of the identifying number of that markpoint.

To rollback to an earlier state in the process, use the **POM_roll_to_markpoint** function specifying the markpoint to which you want to rollback. This function restores the local memory and the database to the corresponding state that it was in when the specified markpoint was set.

As an overhead saving, rollback can be turned off with the **POM_set_env_info** function.

The rollback removes the caller's changes to the database. Any changes to the database by other users are unaffected. If other users have made changes which prevent the caller's changes being reversed (for example, deleted or changed values in an instance in the database), then the rollback fails.

Time-outs

You can use the **POM_set_timeout** function to set a session wide time-out value to determine the period during which a POM function repeatedly tries to lock an instance for change (for example, when calling the **POM_load_instances** function).

After the time-out period, if the POM function that is to perform the change is unsuccessful at getting the lock, it returns the appropriate failure argument as described in the relevant function.

The **POM_ask_timeout** function advises the user what time-out period has been set for the session in seconds.

Errors

In the event of a failure, the **POM_explain_last_error** function explains why the last failing POM function failed. The information that is returned by this function includes:

- The actual error code.

- The argument that caused the function to fail.

- The name of the function that failed.

- A description of what that particular error code means.

The **POM_describe_error** function returns a string describing the specified error code.

The description returned by this function is the same as that returned by **POM_explain_last_error**. For convenience the **POM_describe_error** function can be called with any error code at any time.

Control of logons

The control of logons, which allows or prevents other users from accessing POM, is set by the **POM_set_logins** function. When the system administrator needs to perform an operation that requires only one POM user, the system administrator can use the **POM_set_logins** function to prevent any other user from logging into the POM. In the unusual event of the process which disabled logons terminating abnormally, the system administrator does have the authority to log on using the appropriate group and password as identification, thus overriding the effect of the earlier **POM_set_logins** function call.

However, all POM users can use the **POM_ask_logins** function to determine whether other users are currently allowed to access the POM.

Environmental information

The **POM_set_env_info** function offers alternative methods of enabling and disabling operations as explained in its description. This function sets the following environmental information:

- Enable or disable rollback.

- Enable or disable attribute value checking in the local memory (for example, for the duplication of unique attributes or for checking against upper/lower bounds).

- Enable or disable journalling (for example, for writing a function's arguments to the system log when debugging and bug reporting).

- Enable or disable argument checking.

- Log on of a traceback to the log file when receiving an error.

- Writing of all of POM SQL to the log file.

The **POM_ask_env_info** function returns the environmental information that was specified by the **POM_set_env_info** function.

SQL

The application data stored by the POM module is accessible using direct SQL statements for read and write purposes. The **POM_sql_view_of_class** function creates a database view of the selected attributes in the given class. The view is given the name supplied and the columns are aliased with the column names that are given in the argument for the column names.

The attributes can be from a specified class or any superclass, but they cannot include any array types.

Direct SQL access to the internal structure

These functions have been provided to allow the end user access to the database tables used by the POM:

- The **POM_dbname_of_att** function takes a class identifier and an attribute identifier and returns the textual names of the SQL database table and the column within that table in which the relevant attribute is stored. This function can be used to either directly access the values of a specified attribute for specific instances or to produce a query over all values of the relevant attribute in a given class.

- The **POM_type_of_att** function returns the array type of the attribute identified by its **class_id** and **attr_id**. The returned types of arrays are:

  - Nonarray (for example, an integer)

  - Small array

  - Large array

  - Variable length array

- The **POM_tag_to_uid** function returns a UID, which is an internal reference to an instance.

POM functions available for user assistance

Because some POM tables (in other words, those that hold the values for the class variables) have the entries specified by a combination of the site identifier, internal attribute, and class identifiers, you must use the following functions:

- **POM_site_id**

  Returns the site ID of the local POM installation.

- **POM_attr_to_apid**

  Converts the external class and attribute identifiers into the internal integer attribute identifier.

- **POM_class_to_cpid**

  Converts the external class identifier into the internal integer representation for a class.

- **POM_get_char_ordering**

  Returns the ordering on the character set in use. This is useful when creating an enquiry which selects on a string-valued attribute.

- **POM_describe_token**

  Returns a string describing the specified token (for example, for the **POM_int** token, which is a defined value, the returned string might be integer).

- **POM_free**

  Calls **MEM_free**. Frees the space that the POM allocated internally when returning an array of information.

Values that can be handed to this function are indicated by the letters **<OF>** after the argument in the function syntax.

POM enquiry module

Introduction to the POM enquiry module

With the POM enquiry system, you can ask the database directly for information instead of navigating loaded objects. For example, you can request a list of all the workspace objects where the **object_name** is **test**. Another example is a list of all the instances of a class that have a reference to a particular tag. If you use normal ITK objects to try to get this information, it takes a lot of time and memory. The POM enquiry system gives you this information much more quickly and efficiently. The enquiry system allows

you to develop queries that conform to the SQL-92 standard with a few exceptions. These exceptions are:

- You cannot use **SELECT * FROM**. Only named attributes are allowed.

- You cannot use Cartesian products.

- You cannot use outer joins.

> Note:
>
> POM enquiries are not the same as saved queries. They are completely different mechanisms.

Differences between old and new POM enquiry module systems

There are actually two different enquiry systems—a newer one and an older one. You should use the newer one with the calls that begin with **POM_enquiry_** because:

- It produces better, more efficient SQL which runs faster.

- It supports more features present in SQL and allows you to do more complex queries.

- You can ask for more than one attribute to be returned. The old system only returned the tags of matching objects.

- You can call the **POM_load_instances_possible_by_enquiry** function using a new enquiry and have all the matching instances loaded efficiently.

The definition of all the API calls for the new enquiry system are in the **enq.h** header file and the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

If you have legacy code, you can continue to use the old enquiry system. However, only the newer enquiry system is described here. To make the code more readable, the return values of the API calls are ignored in the examples. In ITK, you should always check return values before proceeding to the next call.

Starting to use the POM enquiry module

POM enquiry module basic concepts

There are a number of basic building blocks for an enquiry. These are described below and then the examples explain what you actually do with these and what API calls to use.

Each enquiry has a unique name used to identify it. You specify the name when you create the enquiry and must specify this name for anything you do with this enquiry. Once you create an enquiry, it exists in memory until you delete it. Therefore, you can access it using the unique name even if the tag for it has gone out of scope. It also means you must clean it up explicitly once you've finished with it. An enquiry is made up of a number of smaller parts:

- **SELECT attributes**

  The values you want to get back from the enquiry.

- **ORDER BY clauses**

  If you want your results to come back in a particular order, you can use an **ORDER BY**. Note that this adds a **SELECT** attribute for the thing you are ordering by.

- **Attribute expressions**

  These allow you to control what data you want returned. These expressions form the **where** part of the generated SQL statement. There are two main types of attributes:

  - **Fixed attributes**

    These are the same every time you run the enquiry. They can be simple values (for example, **name = 'Bill'**) or used to create joins between tables (described below).

  - **Bind variables**

    These allow different values to be passed to the database each time the enquiry is run. These typically would be used either to find related objects or for values that you do not know at compile time (for example, the value of tags).

  When you add attribute expressions, you normally give both the name of the attribute and the class it is from.

- **Where clause**

  This combines a number of attribute expressions using different set expressions. The most common one is **AND**.

These parts are the ones that you come across most often. There are a few more than can be used; they are listed later in this document.

For example, compare the different building blocks with a very simple SQL statement:

```
SELECT attr1 from table1 where attr3='test' ORDER BY attr1
```

- **attr1** is a **SELECT** attribute

- **where attr3='test'** is a where clause and

- **attr3** and **'test'** are attribute expressions

- **ORDER BY attr1** is an **ORDER BY** clause

Basic steps to creating POM enqueries

If you create a POM enquiry, you must follow a number of steps and answer a number of questions:

1. Create your query. Each query must have a unique name; it should be something that you can recognize later and describes what it does.

2. Add the data you want returned to your enquiry.

3. Add any attributes you need for where clauses, joins, or other items.

4. Create expressions with the attributes from the previous step to make a where clause.

5. Optionally, add an **ORDER BY** clause.

6. Execute the enquiry and extract the results.

7. Clean up.

Below are a number of examples that illustrate these steps for different cases.

Basic single table enquiry

This is a very simple case. This example finds all the workspace objects whose type is equal to a string that is passed in. The following code shows a single bind variable to compare strings:

```
/* Create enquiry and give it a name */
POM_enquiry_create ("find_wso_by_type");

/* Add select attributes. We want the uids of the workspace
objects so we can load them: */
char * select_attrs[1] = {"puid"};
POM_enquiry_add_select_attrs ("find_wso_by_type",
     "workspaceobject", 1, &select_attrs);

/* Add any attribute expressions. We need two of them – one
for the value we are given and another for the attribute we are
testing against: */
POM_enquiry_set_string_value ("find_wso_by_type", "test",
```

```
       1,&name, POM_enquiry_bind_value );
POM_enquiry_set_attr_expr ("find_wso_by_type", "expr1",
       "workspaceobject", "attr3",
       POM_enquiry_equal, "test" );

/* Set the where condition so we only get back matching rows: */
POM_enquiry_set_where_expr ("find_wso_by_type", "expr1");

/* Now execute the query: */
POM_enquiry_execute ("find_wso_by_type", &n_rows,  &n_cols,
       &values);
```

> Note:
>
> In the example, **expr1** is any string.

The program can then iterate through the results and copy the results. Once that is done, the results array should be freed up using the **MEM_free** function.

Classes and attributes in POM queries

POM queries are largely built in terms of class and attribute names. Where a class name is required, you can use the name of any POM class or any persistent Teamcenter type. (POM enquiry does not support queries on run-time Teamcenter types.) On execution, the query returns results from the instances of the class or type you use. You do not need to know that names of the underlying database tables that POM uses to store class or type instances.

An attribute is always associated with a class. Where an attribute name is required, you can use the name of any of the class's attributes or any of its persistent compound properties. (There are some cases where compound properties are not supported.) You do not need to know the names of the underlying database columns that POM uses to store the attributes.

For example, if you want to get the **object_desc** attribute of an item, you would refer to it using the **Item** class and **object_desc** attribute. It does not matter that the **object_desc** attribute is actually on the **workspaceobject** parent class or what the name of the column the data is stored in. The one exception is the **puid** attribute. This is, in effect, the tag of the object and its unique identifier. If you want to load an object or follow a reference from one object to another, you must use this attribute.

Compound properties and POM enquiry

POM enquiry supports queries on both attributes and on certain compound properties. Whether a compound property is supported depends on how it is defined. POM enquiry only supports compound properties:

- Whose values are stored in the database (not those calculated at run time).

- That are defined in terms of traversing POM references or Teamcenter relations (not those defined in terms of traversing from an item to its item revision).

- That are defined in terms of traversing to objects whose type is specified (rather than in terms of traversing untyped references).

- If the compound property value comes from a form, where the class used to store the value is the one specified in the current property definition (rather than some other class specified in a previous version of the definition).

Housekeeping for POM enquiries

Once a POM enquiry has been created, it is available until it is deleted with the **POM_enquiry_delete** function. Therefore, you can either:

- Create the enquiry every time you use it and then delete it immediately afterwards, or

- Create it once and reuse it during the session.
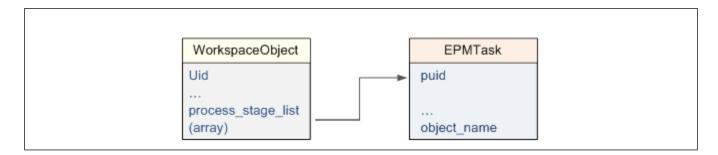
Reusing an enquiry is more efficient. However, be careful in case your code is called just before a **POM_roll_to_markpoint** function call or something else that undoes any allocations in the POM. To protect yourself against this, use a static variable and the **POM_cache_for_session** function. For example:

```
static tag_t my_tag = null_tag;
if (my_tag == null_tag)
{
        /* Create enquiry objects, set up bind values, etc */
        …
        POM_cache_for_session( &my_tag );
        my_tag = 1;
}
else
{
        /* just set up bind values */
        …
}
/* now execute the enquiry and use the results */
…
```

The **POM_cache_for_session** function adds to the variable passed to it the data that is rolled back during an UNDO operation or the **POM_roll_to_markpoint** function. If one of these occurs, then your variable is set back to its previous value (in this case **null_tag**) and you know you have to create the enquiry again.

How to do a simple join for POM enquiry

A join is when you link one table or object to another. For this example, use entries in the **Process Stage List** array and use them to join to EPM tasks. The following figure shows how they relate to each other.

Simple join

Each entry in the **process_stage_list** array points to an **EPMTask** using its **puid** field. The example in the following code is more complicated. The information you want is the **object_name** attribute on all the **EPMTask**s referenced by the workspace object:

```
/* Create enquiry and give it a name */
POM_enquiry_create ("find_referenced_task");

/* Add select attributes. We want the object_name of the Task: */
char * select_attrs[1] = {"object_name"};
POM_enquiry_add_select_attrs ("find_referenced_task",
  "EPMTask", 1, &select_attrs);

/* We're starting from the workspace object, so we must add a
couple of attribute expressions for that:*/
/* This is going to be the tag we have for the WSO. */
POM_enquiry_set_tag_value ("find_referenced_task" , "wso_tag" , 1,
  &object, POM_enquiry_bind_value )
POM_enquiry_set_attr_expr ("find_referenced_task" , "this_wso",\
  "workspaceobject" , "puid" , POM_enquiry_equal , "wso_tag" )

/* And now join the workspace object to the EPMTasks via the
process_stage_list attribute on the workspace object */
POM_enquiry_set_join_expr ("find_referenced_task" , "task_join" ,
    "workspaceobject" , "process_stage_list" ,
    POM_enquiry_equal , "EPMTask" , "puid" )

/* All that's left now is to execute the query: */
POM_enquiry_execute ("find_wso_by_type", &n_rows,
  &n_cols, &values);
```

Advanced use of the POM enquiry module

POM enquiry module advanced concepts

To fully understand and be able to create queries, you must understand the following important terms and concepts:

Class alias

> This is the equivalent of a table alias in the SQL-92 standard. The class alias is very useful if you want to define a query that has a self-join. The classic example of this is employees and managers (Find the list of managers and their employees).

Pseudo class

> The pseudo class represents a compound attribute of a given Teamcenter class. The attribute must be either a:

> - Large array (LA) or

> - Variable length array (VLA)

> This is useful if you want to query the LA or VLA based on a condition on one of the array attribute.

Set expression

> This is the way of combining the result of two or more queries into one using (UNION,...). The queries must be union-compatible (in other words, they must have the same number of attributes selected and they must respectively be of the same domain).

Expression

> There are two types of expressions that can be built in an SQL statement. These are:

> - **Logical expressions**

>   Any expression that results to a logical value (true or false). For example, A or B, (A and B) or C, A=B, A>=B, A IS NULL, A in [a,b,c].

> - **Function expressions**

>   Any expression that uses a function to manipulate an attribute. For example, SUBSTR (A,1,14), UPPER, A+B,LOWER(A), MAX(A), MIN(A), A+B,A/B.

Escape character

> Teamcenter provides two special wildcard characters: **POM_wildcard_character_one** and **POM_wildcard_character_any**. You can set these values by calling the **POM_set_env_info** function. However, if the string contains any of these wildcard characters as part of the data, you have no way of telling Teamcenter how to treat these values. For this reason, there is a token for the escape character called **POM_escape_character**.

You can set this character in the same way as for the **POM_wildcard_character_any** or **POM_wildcard_character_one** characters, by calling the **POM_set_env_info** function with its first argument **POM_escape_character** token. If any of the wildcard characters is part of the data, then you must escape it with the chosen escape character.

> Note:
>
> • If the escape character is found in the string and is not followed by either the wildcard characters or itself, the POM removes it from the string.
>
> • The **PUID**, **PSEQ** and **PVAL** keywords are reserved Teamcenter words.
>
>   **PUID** can be used as an attribute of any POM class. **PSEQ** and **PVAL** can only be used as attributes of a pseudo-class.

Query specification

The SQL-92 standard query specification consists of the following clauses:

• **SELECT** *select-list*

• **FROM** *class-list*

• [**WHERE** *search-condition*]

• [**ORDER BY** *attribute-list*]

• [**GROUP BY** *attribute-list*]

• [**HAVING** *search-condition*]

Clauses enclosed by brackets [ ] are optional.

Building expressions

The POM enquiry system APIs contain a list of functions that can be used to build expressions.

Function	Description
POM_enquiry_set_attr_expr	This is the recommended way for creating expressions on **class.attr**. It is used as follows:
	Create a type value: **POM_enquiry_set_type_value (***enqid***, ***valid***, ***n_values***, ***values***, ***property***)**; here *property* can be either **POM_enquiry_const_value** or **POM_enquiry_bind_value**.

Function	Description
	Create the expression: **POM_enquiry_set_attr_expr (***enqid***, ***exprid***, ***class***, ***attr***, ***operator***, ***valid***)**
	Note: *valid* cannot be **NULL**. If the right-hand side of the expression is **NULL** such as for the unary operators **UPPER** and **LOWER**, you must create (**char *valid=""**).
POM_enquiry_set_type_expr	This is a convenient function that avoids the need to create a value. It is used as follows: **POM_enquiry_set_type_expr (***enqid***, ***exprid***, ***class***, ***attr***, ***operator***, ***type***, ***value***)** *type* must be the type of the attribute in question. *type* can be: int, double, char, string, logical, **tag_t**, **date_t**. In the case of **tag_t** and **date_t**, the function name is **POM_enquiry_set_tag_expr** and **POM_enquiry_set_date_expr**. However, this function cannot be used to create expressions on **class.attr** where the class has no associated database table (such **POM_system_class**). Use the **POM_enquiry_set_attr_expr** function instead.
POM_enquiry_set_expr	This function is used to combine two expressions into one. For example, **- expr1 OR expr2, expr1 AND expr2. - UPPER (SUBSTR (A, 1,14)). - expr1** *theta-op* **expr2** where *theta-op* can be: **=,>,>=,<,<=**.... **(a+b)*c.** Note: If you have an expression like the following: **(class.attr1 + class.attr2)/class.attr3** in the select-clause, you need to add an expression to the where-clause as follows: **((class.attr3 > 0) AND (class.attr3 IS NOT NULL))**.

Practical example

For example, if you want to find the name and type of all objects created since March 8, 2005, the SQL statement is:

```
SELECT object_name, object_type FROM WorkspaceObject WHERE creation_date
>=
08-Mar-2005 00:00
```

The POM enquiry APIs required to create this query are shown in the following code:

```
void     ***report;
int        n_rows, n_cols, row, column;
const char    *select_attr_list[] = { "object_name", "object_type"};
date_t aDate;

ITK_string_to_date("08-Mar-2005 00:00", &aDate);
/*create a query*/
POM_enquiry_create("aEnqId");
/*add the list to the select clause of the query*/
POM_enquiry_add_select_attrs("aEnqId", "WorkspaceObject", 2,
select_attr_list);
/*create a date value object.*/
POM_enquiry_set_date_value("aEnqId", "aValId", 1, &aDate,
POM_enquiry_bind_value);
/* create an expression for pcreation_date = Date using the value "aValId"
*/
POM_enquiry_set_attr_expr("aEnqId", "aExpId", "WorkspaceObject",
    "creation_date", POM_enquiry_greater_than_or_eq, "aValId");
/*set the where clause search condition.*/
POM_enquiry_set_where_expr( "aEnqId", "aExpId"));
/*execute the query*/
POM_enquiry_execute("aEnqId", &n_rows, &n_cols, &report);
for (row = 0; row < n_rows; row++)
{
    for (column = 0; column < n_cols; column++)
    printf("%s\t", report[row][column]);
    printf("\n");
}
```

Data in the report variable can be read as **report[*i*][*j*]** where *i* is the row and *j* is the column.

As can be seen from the previous example, the **FROM** clause was not defined. It is automatically derived from the other clauses of the query.

SELECT clause

The **SELECT** list of an SQL statement consists of a list of attributes and expressions. The operator of the expression must be a valid one supported by the RDBMS in use. The following are supported: **+, -, /, *, substr, upper, lower, contact, to_number, to_date, cpid_of, uid_of, max, min, avg, count all, count distinct,** and **sum**.

Refer to the **Supported operators** table for valid token names supported by the enquiry system.

For example, if you want to find all the objects created by the users Ahmed, Hewat, and James, the SQL statement would be:

```
    SELECT pa.object_name,pr.user_name FROM pom_application_object pa, user
    ur,
    person pr WHERE ur.person = pr.puid and pr.puid = pa.puid and
    pr.user_name in ('Ahmed','Hewat','James')
```

The POM enquiry APIs required to create this query are as follows:

As can be seen from the previous SQL statement, the **object_name** and **user_name** attributes do not belong to the same class. Therefore you need to call the **POM_enquiry_add_select_attrs** function twice, as shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/*create two variables called select_attr_list1 and select_attr_list2 */
const char * select_attr_list1[] = { "object_name"};
const char * select_attr_list2[] = { "user_name" };
/*create a variable value_list */
const char * value_list[] = {"Ahmed","Hewat", "James"};
POM_enquiry_add_select_attrs ( "a_unique_query_id",
"pom_application_object", 1,
select_attr_list1 );
POM_enquiry_add_select_attrs ( "a_unique_query_id", "person", 1,
select_attr_list2
);
/*Create the join expr ur.rpersonu = pr.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_1","user","person",
POM_enquiry_equal,"person","puid"
);
/*Create the join expr pr.puid = pa.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_2","person","puid",
POM_enquiry_equal,"pom_application_object","puid"
);
/*Create a tag value.*/
POM_enquiry_set_string_value ( "a_unique_query_id", "aunique_value_id", 3,
value_list,
POM_enquiry_bind_value );
/*Create the expression user_name in ('Ahmed','Hewat','James').*/
POM_enquiry_create_attr_expr ( "a_unique_query_id","auniqueExprId_3",
"person",
"user_name",
POM_enquiry_in, "aunique_value_id" );
/*Combine the expr_1 and expr_2 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_4",
"auniqueExprId_1",
```

```
POM_enquiry_and, "auniqueExprId_2" );
/*Combine the expr_3 and expr_4 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_5",
"auniqueExprId_3",
POM_enquiry_and, "auniqueExprId_4" );
/*Set the where clause of the query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_5" );
/*Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

Notice that the **POM_enquiry_bind_value** token specifies that the query can be rerun with different values without the need to re-evaluate the query. Also by using bind variables, you are telling the database server to cache the plan of the execution of the SQL statement which saves time.

FROM clause

The **FROM** clause of the query is automatically generated.

WHERE clause

The **WHERE** clause of a query restricts the result to only those rows that verify the condition supplied.

For example, if you want to find the name and type of all objects created between date1 and date2, the SQL statement is:

```
SELECT object_name,object_type FROM pom_application_object WHERE
creation_date
BETWEEN date1 and date2
```

The POM enquiry APIs required to create this query are shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/*create a query*/
POM_enquiry_create ("a unique query id")
/*create a variable called select-attr_list*/
const char * select_attr_list[] = { "object_name","object_type"};
/*add the list to the select clause of the query*/
POM_enquiry_add_select_attrs ("a unique query id" ,
"pom_application_object", 2 ,
select_attr_list );
/*create two date_t variables.*/
const date_t aDate1;
const date_t aDate2;
const date_t date_list[2];
```

```
/*initialize adate1. */
aDate1.day = day1; aDate1.month = month1; aDate1.year = year1; aDate1.hours
= 0;
aDate1.minutes = 0;
aDate1.seconds = 0;
/*initialize adate2. */
aDate2.day = day2; aDate2.month = month2; aDate2.year = year2; aDate2.hours
= 0;
aDate2.minutes = 0;
aDate2.seconds = 0;
date_list[0]=aDate1;
date_list[1]=aDate2;
/*create a date value object.*/
POM_enquiry_set_date_value ("a unique query id" , "auniquevalueId", 2 ,
date_list
, POM_enquiry_bind_value );
/*create an expression for pcreation_date between aDate1 and aDate2 using
the value
"auniquevalueId"*/
POM_enquiry_set_attr_expr ("a unique query id","auniqueExprId",
"pom_application_object","creation_date",POM_enquiry_between,
"auniquevalueId");
/*set the where clause search condition.*/
POM_enquiry_set_where_expr ( "a unique query id","auniqueExprId");
/*execute the query*/
POM_enquiry_execute ( "a unique query id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

ORDER BY clause

The **ORDER BY** list of an SQL statement consists of a list of attributes and expressions. The operator of the expression must be a valid one supported by the RDBMS in use.

Refer to the Supported operators table for valid token names supported by the enquiry system.

For example, if you want to find all the objects created by the users Ahmed, Hewat and James ordered by the **user_name** attribute, the SQL statement is:

```
SELECT pa.pobject_name,pr.user_name FROM ppom_application_object pa,
puser ur,
pperson pr WHERE ur.rpersonu = pr.puid and pr.puid = pa.puid and
pr.puser_name
in ('Ahmed','Hewat','James') ORDER BY user_name
```

As can be seen from the previous SQL statement, the **object_name** and **user_name** attributes do not belong to the same class, therefore you need to call the **POM_enquiry_add_select_attrs** function twice. The POM enquiry APIs required to create this query are shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/*create two variables called select_attr_list1 and select_attr_list2 */
const char * select_attr_list1[] = { "object_name"};
const char * select_attr_list2[] = { "user_name" };
/*create a variable value_list */
const char * value_list[] = {"Ahmed","Hewat", "James"};
POM_enquiry_add_select_attrs ( "a_unique_query_id",
"pom_application_object", 1,
select_attr_list1 );
POM_enquiry_add_select_attrs ( "a_unique_query_id", "person", 1,
select_attr_list2
);
/*Create the join expr ur.rpersonu = pr.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_1","user","person",
POM_enquiry_equal,"person","puid"
);
/*Create the join expr pr.puid = pa.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_2","person","puid",
POM_enquiry_equal,"pom_application_object","puid"
);
/*Create a tag value.*/
POM_enquiry_set_string_value ( "a_unique_query_id", "aunique_value_id", 3,
value_list,
POM_enquiry_bind_value );
/*Create the expression user_name in ('Ahmed','Hewat','James').*/
POM_enquiry_create_attr_expr ( "a_unique_query_id","auniqueExprId_3",
"person",
"user_name",
POM_enquiry_in, "aunique_value_id" );
/*Combine the expr_1 and expr_2 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_4",
"auniqueExprId_1",
POM_enquiry_and, "auniqueExprId_2" );
/*Combine the expr_3 and expr_4 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_5",
"auniqueExprId_3",
POM_enquiry_and, "auniqueExprId_4" );
/*Set the where clause of the query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_5" );
/*set the order by clause.*/
POM_enquiry_add_order_attr ( "a_unique_query_id","Person","user_name",
POM_enquiry_asc_order
);
```

```
/*Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

GROUP BY clause

This the same as the **ORDER BY** clause except for the sorting order. Just replace the **POM_enquiry_add_order_attr** function with the **POM_enquiry_add_group_attr** function and the **POM_enquiry_add_order_expr** function with the **POM_enquiry_add_group_expr** function.

HAVING clause

This is the same as the **WHERE** clause. Replace the **POM_enquiry_set_where_expr** function with the **POM_enquiry_set_having_expr** function.

Subqueries

For example, if you want to find the list of folders whose contents attribute contains a **UGMASTER** type dataset, the SQL statement is:

```
SELECT f1.puid FROM folder f1 WHERE f1.contents in ( SELECT ds.puid FROM
dataset ds WHERE ds.object_type = 'UGMASTER' );
```

The POM enquiry APIs required to create this query are shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/*create a variable select_attr_list */
const char * select_attr_list[] = {"puid"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
As can be seen from the above pseudo-SQL statement contains a subquery.
Hence the need for creating one.
/*Create a subquery.*/
POM_enquiry_set_sub_enquiry ( "a_unique_query_id", "subquery_unique_id" );
/* select the puid from Folder class.*/
POM_enquiry_add_select_attrs ( "a_unique_query_id", "Folder", 1,
select_attr_list
);
/* select the puid from Dataset class.*/
POM_enquiry_add_select_attrs ( "subquery_unique_id", "Dataset", 1,
select_attr_list
);
/*Create the expr ds.type = 'UGMASTER'.*/
POM_enquiry_set_string_expr
("subquery_unique_id","auniqueExprId_1","Dataset",
```

```
"object_type",POM_enquiry_equal,"UGMASTER"
);
/*Set the subquery where clause.*/
POM_enquiry_set_where_expr ( "subquery_unique_id","auniqueExprId_1" );
/*Create the expr f1.contents in subquery.*/
POM_enquiry_set_attr_expr
("a_unique_query_id","auniqueExprId_2","Folder","contents",
POM_enquiry_in,"subquery_unique_id"
);
/*Set the outer query where clause.*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_7" );
/*Execute the outer query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

> Note:
>
> The **POM_enquiry_in** function can be replaced by the **POM_enquiry_not_in**, **POM_enquiry_exists**, or **POM_enquiry_not_exists** function.

For **IN**, **NOT IN**, **EXISTS** and **NOT EXISTS** operators, see your RDBMS reference manual.

Use of class_alias

For example, if you want to find the list of folders whose contents attribute contains a folder of type 'override' and its contents has an item revision X, the pseudo-SQL is:

```
select f1.puid from folder f1,folder f2, item_revision itr where
f1.contents
= f2.puid and f2.type = 'override' and f2.contents = itr.puid and
itr.object_name = 'X';
```

The folder class appears twice in the from clause and therefore needs a class alias.

The POM enquiry APIs required to create this query are shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/*create a variable select_attr_list */
const char * select_attr_list[] = {"puid"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/*Create a class alias for the Folder class.*/
POM_enquiry_create_class_alias ( "a_unique_query_id",
"Folder",1,"Folder_alias");
/* select the puid from Folder class.*/
```

```
POM_enquiry_add_select_attrs ( "a_unique_query_id", "Folder", 1,
select_attr_list
);
/*Create the join expr f1.contents = f2.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_1","Folder","contents",
POM_enquiry_equal,"Folder_alias","puid"
);
/*Create the expr f2.type = 'override'.*/
POM_enquiry_set_string_expr
("a_unique_query_id","auniqueExprId_2","Folder_alias",
"type",POM_enquiry_equal,"override"
);
/*Create the join expr f2.contents = itr.puid*/
POM_enquiry_set_join_expr
("a_unique_query_id","auniqueExprId_3","Folder_alias",
"contents",POM_enquiry_equal,"Item_revision","puid"
);
/*Create the expr itr.object_name = 'X'.*/
POM_enquiry_set_string_expr
("a_unique_query_id","auniqueExprId_4","Folder_alias",
"object_name",POM_enquiry_equal,"X"
);
/*Combine the expr_1 and expr_2 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_5",
"auniqueExprId_1",
POM_enquiry_and, "auniqueExprId_2" );
/*Combine the expr_3 and expr_4 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_6",
"auniqueExprId_3",
POM_enquiry_and, "auniqueExprId_4" );
/*Combine the expr_5 and expr_6 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_7",
"auniqueExprId_5",
POM_enquiry_and, "auniqueExprId_6" );
/*Set the where clause of the query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_7" );
/*Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

Use of pseudo_class

For example, if you want to find all versions of the dataset 'X', the pseudo-SQL is:

```
select ds.puid from dataset ds, revisionAnchor rva where
ds.rev_chain_anchor
= rva.puid and rva.revisions.PSEQ = 0 and ds.object_name = 'X'
```

The **PSEQ** attribute in the where clause is not a **POM_attribute** and the 'revisions' is not a **POM_class**. Since the only classes you are allowed to query on are **POM_classes**, you have to convert the revisions attribute of the **revisionAnchor** into a **POM_class**. To do this, create a pseudo-class on the attribute revisions of the **revisionAnchor**.

> Note:
>
> You can only create pseudo-classes for VLA or LA attributes.

The POM enquiry APIs required to create this query are shown in the following code:

```
/* Create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/* Create a variable select_attr_list */
const char * select_attr_list[] = {"puid"};
/* Create a variable int_val=0*/
const int int_val = 0;
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/* Create a pseudo-class for the revisionAnchor.revisions attribute.*/
POM_enquiry_set_pseudo_calias ( "a_unique_query_id",
"RevisionAnchor","revisions",
"class_revisions");
/* Select the puid from Dataset class.*/
POM_enquiry_add_select_attrs ( "a_unique_query_id", "Dataset", 1,
select_attr_list);
/* Create the expr rva.revisons.pseq = 0.*/
/* First create an int value.*/
POM_enquiry_set_int_value ( "a_unique_query_id","auniqueValueId", 1,
&int_val,
POM_enquiry_bind_value
);
POM_enquiry_set_attr_expr
("a_unique_query_id","auniqueExprId_1","class_revisions",
"pseq",POM_enquiry_equal,"auniqueValueId"
);
/* NOTICE: the 'pseq' attribute is reserved keyword in TC/POM, it can be
used as an
attribute of the pseudo-class. The same
apply to pval and puid. */
/* Create the expr ds.object_name = 'X'.*/
POM_enquiry_set_string_expr
("a_unique_query_id","auniqueExprId_2","Dataset",
"object_name",POM_enquiry_equal,"X"
);
/* Create the join expr ds.rev_chain_anchor = rva.puid */
POM_enquiry_set_join_expr ("a_unique_query_id","auniqueExprId_3","Dataset",
```

```
"rev_chain_anchor",POM_enquiry_equal,"revisionAnchor","puid");
/* Combine the expr_1 and expr_2 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_4",
"auniqueExprId_1",
POM_enquiry_and, "auniqueExprId_2" );
/* Combine the expr_3 and expr_4 using AND.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_5",
"auniqueExprId_3",
POM_enquiry_and, "auniqueExprId_4" );
/* Set the where clause of the query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_5" );
/* Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/* Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

Use of Set-Expression

INTERSECTION

For example, if you want to find all the objects that are common to two folders, the pseudo SQL is as follows:

```
select f1.contents from folder f1 where f1.object_name='Compare'
INTERSECTION
select f2.contents from folder f2 where f2.object_name='Reference'
```

The pseudo SQL consists of two queries that are combined by the **INTERSECTION** set operator. In this case, one of the queries must be the outer query and the other one must be scoped to it. Therefore, the non-outer query must be created as a subquery of the outer one.

The POM enquiry APIs required to create this query are shown in the following code:

```
/* Create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/* Create a variable select_attr_list */
const char * select_attr_list[] = {"contents"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/* Select the contents from Folder class.*/
POM_enquiry_add_select_attrs ( "a_unique_query_id", "Folder", 1,
select_attr_list );
/* Now limit it to just the folder we want to see */
/* Folder names are not always unique so for a real example you should */
/* use the tag of the folder */
POM_enquiry_set_string_expr ("a_unique_query_id","auniqueExprId_1","Folder",
 "object_name", POM_enquiry_equal, "Compare" );
```

```
/* Set the outer query where clause.*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_1" );
/* POM implements set queries as subqueries and so we need to create one */
POM_enquiry_set_sub_enquiry ( "a_unique_query_id", "subquery_unique_id" );
/* Create a class alias for the Folder class. The folder class appears
twice - once */
/* in the outer query and the other time in the subquery. */
POM_enquiry_create_class_alias ( "subquery_unique_id",
"Folder",1,"Folder_alias");
/* Select the contents from other folder.*/
POM_enquiry_add_select_attrs ( "subquery_unique_id", "Folder_alias", 1,
select_attr_list);
POM_enquiry_set_string_expr
("a_unique_query_id","auniqueExprId_2","Folder_alias",
"object_name", POM_enquiry_equal, "Reference" );
/* Set the subquery where clause.*/
POM_enquiry_set_where_expr ( "subquery_unique_id","auniqueExprId_2" );
/* Add the set expression */
POM_enquiry_set_setexpr ("a_unique_query_id", POM_enquiry_intersection,
      "subquery_unique_id");
/* Execute the outer query.*/
POM_enquiry_execute ( "a_unique_query_id",&row, &cols, &report);
/* Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

DIFFERENCE

For example, if you want to find all the objects that are in a reference folder that do not exist in a named folder, the pseudo SQL is:

```
select f1.contents from folder f1 where f1.object_name='Reference'
DIFFERENCE
select f2.contents from folder f2 where f2.object_name='Compare'
```

The pseudo SQL consists of two queries that are combined by the **DIFFERENCE** set operator. In this case, one of the queries must be the outer query and the other one must be scoped to it. Therefore, the non-outer query must be created as a subquery of the outer one.

The POM enquiry APIs required to create this query are shown in the following code:

```
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/*create a variable select_attr_list */
const char * select_attr_list[] = {"contents"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/* select the contents from Folder class.*/
```

```
POM_enquiry_add_select_attrs ( "a_unique_query_id", "Folder", 1,
select_attr_list );
/* Now limit it to just the folder we want to see */
/* Folder names are not always unique so for a real example you should */
/* use the tag of the folder */
POM_enquiry_set_string_expr ("a_unique_query_id","auniqueExprId_1","Folder",
 "object_name", POM_enquiry_equal, "Reference" );
/*Set the outer query where clause.*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_1" );
/* POM implements set queries as subqueries and so we need to create one */
POM_enquiry_set_sub_enquiry ( "a_unique_query_id", "subquery_unique_id" );
/* Create a class alias for the Folder class. The folder class appears
twice - once */
/* in the outer query and the other time in the subquery. */
POM_enquiry_create_class_alias ( "subquery_unique_id",
"Folder",1,"Folder_alias");
/* select the contents from other folder.*/
POM_enquiry_add_select_attrs ( "subquery_unique_id", "Folder_alias", 1,
 select_attr_list);
POM_enquiry_set_string_expr
("a_unique_query_id","auniqueExprId_2","Folder_alias",
"object_name", POM_enquiry_equal, "Compare" );
/*Set the subquery where clause.*/
POM_enquiry_set_where_expr ( "subquery_unique_id","auniqueExprId_2" );
/* Add the set expression */
POM_enquiry_set_setexpr ("a_unique_query_id", POM_enquiry_difference,
 "subquery_unique_id");
/*Execute the outer query.*/
POM_enquiry_execute ( "a_unique_query_id", &row, &cols, &report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

UNION

For example, you have two forms: **FORM1** and **FORM2**. **FORM1** is an engineering type form that has two attributes: **weight** and **material**. **FORM2** is a production type form and has two attributes: **cost** and **effectivity_date**.

If you want to find all engineering type objects that weigh more than **wt1** and all production type objects that cost less than **cost1**, the pseudo SQL is:

```
Select f1.puid From FORM1 Where f1.type = 'engineering' And f1.weight >
wt1
UNION Select f2.puid From FORM2 Where f2.type = 'Production' And
f2.cost < cost1
```

> Tip:
>
> When using the same class in multiple **FROM** clauses associated with a **UNION**, you will need to use a **class alias**.

The POM enquiry APIs required to create this query are shown in the following code:

```
/* create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/* Create a variable select_attr_list */
const char * select_attr_list[] = {"puid"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/* Select the puid from FORM1 class.*/
POM_enquiry_add_select_attrs ( "a_unique_query_id", "form1", 1,
select_attr_list);
/* Create the expr f1.type = 'engineering'*/
POM_enquiry_set_string_expr ( "a_unique_query_id","auniqueExprId_1","form1",
"object_type", POM_enquiry_equal,"engineering");
/* Create the expr f1.weight > wt1*/
POM_enquiry_set_double_expr
( "a_unique_query_id","auniqueExprId_2","form1","weight",
POM_enquiry_greater_than,wt1);
/* Combine expr1 and expr2 using AND operator.*/
POM_enquiry_set_expr ( "a_unique_query_id", "auniqueExprId_3",
"auniqueExprId_1",
POM_enquiry_and,"auniqueExprId_2" );
/* Set the where clause of the outer query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_3" );
/* Create a subquery*/
POM_enquiry_set_sub_enquiry ("a_unique_query_id", "subquery_unique_id" )
/* Select the puid from FORM2 class.*/
POM_enquiry_add_select_attrs ( "subquery_unique_id", "form2", 1,
select_attr_list
);
/* Create the expr f2.cost <cost1*/
POM_enquiry_set_double_expr
( "subquery_unique_id","auniqueExprId_5","form2",
"cost",POM_enquiry_less_than,cost1 );
/* Combine expr4 and expr5 using AND operator.*/
POM_enquiry_set_expr ( "subquery_unique_id", "auniqueExprId_6",
"auniqueExprId_4",
POM_enquiry_and, "auniqueExprId_5" );
/* Set the where clause of the subquery*/
POM_enquiry_set_where_expr ( "subquery_unique_id","auniqueExprId_6" );
/* Now we need to set the set-expression of the outer query. */
POM_enquiry_set_setexpr ( "a_unique_query_id" ,"setexpr",POM_enquiry_union ,
```

```
"subquery_unique_id" );
/* Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/* Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

POM enquiry module special operators

POM_enquiry_substr

If you want to create the expression **SUBSTR (item.desc,1,5)='blob'**, you can use the enquiry APIs as shown in the following code:

```
/*Create a variable int_list*/
const int int_list[]={1,5};
/*Create a string variable*/
const char *str_list[]={"blob"};

/* First we need to create an int value:*/
POM_enquiry_set_int_value
( "a_unique_query_id","auniqueValueId_1",2,int_list,
                        POM_enquiry_const_value);
/* Create the expression SUBSTR ( item.desc, 1,5) using the int value.*/
POM_enquiry_set_attr_expr ( "a_unique_query_id",
"auniqueExprId_1","item","desc",
                        POM_enquiry_substr,"auniqueValueId_1")
/* Create a string value = 'blob'.*/
POM_enquiry_set_str_value
( "a_unique_query_id","auniqueValueId_2",1,str_list,
                        POM_enquiry_bind_value);
/* Create the expression SUBSTR ( item.desc, 1,5) = 'blob'.*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_2",
"auniqueExprId_1",
                   POM_enquiry_equal, "auniqueValueId_2");
```

POM_enquiry_cpid_of

For example, if you want to find all the dataset objects of a folder, the pseudo SQL is:

```
Select contents From Folder Where cpid_of ( contents ) = dataset cpid
```

The way to implement this query in the old query system is to use a join between dataset and folder contents. To avoid the performance penalty of using the join, you can use the **POM_enquiry_cpid_of** special operator.

> Note:
>
> **POM_enquiry_cpid_of** can only be used with typed or **untyped_reference** type attributes.

```
/*Find the cpid of dataset class by using:*/
POM_class_id_of_class ("dataset",&classid );
POM_class_to_cpid ( classid, &cpid );
/*create variables for POM_enquiry_execute*/
int rows,cols;
void *** report;
/*create a variable select_attr_list */
const char * select_attr_list[] = {"contents"};
/* Create a query*/
POM_enquiry_create ("a_unique_query_id")
/* select the puid from folder class.*/
POM_enquiry_add_select_attrs ( "a_unique_query_id", "folder", 1,
select_attr_list);
/*create an int value*/
POM_enquiry_set_int_value ( "a_unique_query_id", "auniqueValueId_1", 1,
&cpid,
POM_enquiry_bind_value);
/* Create the expression cpid_of( contents ).*/
POM_enquiry_set_attr_expr ( "a_unique_query_id", "auniqueExprId_1",
"folder",
"contents", POM_enquiry_cpid_of, "" );
/*Set the expression cpid_of ( contents ) = dataset cpid.*/
POM_enquiry_set_expr ( "a_unique_query_id",
"auniqueExprId_2","auniqueExprId_1",
POM_enquiry_equal,"auniqueValueId");
/*Set the where clause of the query*/
POM_enquiry_set_where_expr ( "a_unique_query_id","auniqueExprId_2" );
/*Execute the query.*/
POM_enquiry_execute ( "a_unique_query_id",&row,&cols,&report);
/*Delete the query*/
POM_enquiry_delete ( "a_unique_query_id" );
```

POM_enquiry_upper and POM_enquiry_lower

If you want to create the expression **UPPER(item.desc)= UPPER('blob')**, you can use the enquiry APIs as shown in the following code:

```
/*Create a string variable*/
const char *str_list[]={'blob'};
/* Create the expression UPPER( item.desc).*/
POM_enquiry_set_attr_expr ( "a_unique_query_id",
"auniqueExprId_1","item","desc",
POM_enquiry_upper,"")
/* Create a string value = 'blob'.*/
```

```
POM_enquiry_set_int_value
( "a_unique_query_id","auniqueValueId_2",1,str_list,
POM_enquiry_bind_value);
/* Create the expression UPPER( item.desc) = UPPER('blob').*/
POM_enquiry_set_expr ( "a_unique_query_id","auniqueExprId_2",
"auniqueExprId_1",
POM_enquiry_equal, "auniqueValueId_2");
```

> **Note:**
>
> When you apply the upper or lower operator to the left hand side of an expression, the operator is applied automatically to the right hand side of the expression.

POM_enquiry_countdist

If you want to create the expression **COUNT(DISTINCT item.puid)**, you can use the enquiry APIs as shown in the following code:

```
/* Create the expression COUNT(DISTINCT item.desc).*/
POM_enquiry_set_attr_expr ( "a_unique_query_id",
"auniqueExprId_1","item","puid",
POM_enquiry_countdist,"")
```

> **Note:**
>
> You cannot use **count (*)**. Instead use **COUNT (ALL class.puid)** by using the **POM_enquiry_countall**. To return only the number of distinct rows, use **COUNT (DISTINCT class.puid)**, by using the **POM_enquiry_countdist**.

The same apply to MAX, MIN, AVG, and SUM except that these apply to different attributes of the class, not the puid.

POM_enquiry_in and POM_enquiry_not_in

These operators can be used either with a subquery or a list of values. Oracle imposes some restrictions on the number of values allowed in the in-list when using bind variables. These are:

- The number of bind values is limited to 254.

- If you create a query with more than **MAX_BIND_VALUE** bind variables, the new query system converts the bind values into constants and construct a query that can be successfully executed. However, the SQL statement is not cached and therefore risks flooding the System Global Area (SGA).

Supported operators

The following table shows the operators that are supported in the POM enquiry module.

Operator	Left	Right	Where it can be used
POM_enquiry_or	expr	expr	WH
POM_enquiry_and	expr	expr	WH
POM_enquiry_equal	attr/expr	expr/value	WH
POM_enquiry_not_equal	attr/expr	expr/value	WH
POM_enquiry_greater_than	attr/expr	expr/value	WH
POM_enquiry_greater_than_or_eq	attr/expr	expr/value	WH
POM_enquiry_less_than	attr/expr	expr/value	WH
POM_enquiry_less_than_or_eq	attr/expr	expr/value	WH
POM_enquiry_between	attr/expr	expr/value	WH
POM_enquiry_not_between	attr/expr	expr/value	WH
POM_enquiry_in	attr/expr	value/ subquery	WH
POM_enquiry_not_in	attr/expr	value/ subquery	WH
POM_enquiry_exists	attr/expr	subquery	WH
POM_enquiry_not_exists	attr/expr	subquery	WH
POM_enquiry_like	attr/expr	value	WH
POM_enquiry_not_like	attr/expr	value	WH
POM_enquiry_union	query	query	-
POM_enquiry_difference	query	query	-
POM_enquiry_intersection	query	query	-
POM_enquiry_countdist	attr	null	SH
POM_enquiry_countall	attr	null	SH
POM_enquiry_max	attr	null	SH
POM_enquiry_min	attr	null	SH
POM_enquiry_avg	attr	null	SH
POM_enquiry_sum	attr	null	SH
POM_enquiry_plus	attr/expr	attr/expr/ value	SW

Operator	Left	Right	Where it can be used
POM_enquiry_minus	attr/expr	attr/expr/value	SW
POM_enquiry_divide	attr/expr	attr/expr/value	SW
POM_enquiry_multiply	attr/expr	attr/expr/value	SW
POM_enquiry_substr	attr/expr	value	SW
POM_enquiry_upper	attr/expr	null	SW
POM_enquiry_lower	attr/expr	null	SW
POM_enquiry_ascii	attr/expr	null	SW
POM_enquiry_concat	attr/expr		S
POM_enquiry_ltrim	attr/expr	null	SW
POM_enquiry_rtrim	attr/expr	null	SW
POM_enquiry_length	attr/expr	null	SW
POM_enquiry_is_null	attr/expr	null	W
POM_enquiry_is_not_null	attr/expr	null	W
POM_enquiry_uid_of	attr	null	W
POM_enquiry_cpid_of	attr	null	W
POM_enquiry_tonumber	attr	null	W
POM_enquiry_todate	attr	null	W
POM_array_length_equals	attr	null	W
POM_array_length_not_equals	attr	null	W

> Note:
>
> SWH means [S]elect [W]here [H]aving clauses and [-] means the operator is applied to the outer query.

Use of POM load by enquiry

The POM can load the objects returned by a query. This has a number of benefits:
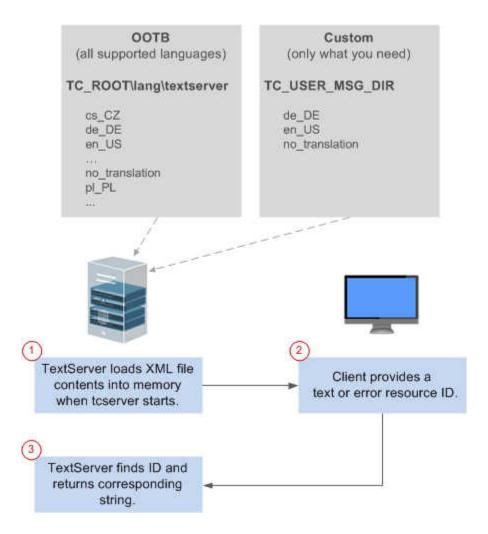
- If you want to load all the objects returned by a query, you can make just one call which both executes the query and loads the objects.

- The loading of objects is more efficient than a normal **POM_load_instances** (or **loadInstances** in **CXPOM**) and results in fewer SQL trips.

To use this functionality:

1. Make sure all the objects you want returned are of the same class. The POM does not support loading objects of different classes using this method.

2. Create the enquiry as described above ensuring you have only one **SELECT** attribute, the **puid** of the class you want to load. You can order the results as long as you are ordering by one of the attributes of the class you are loading.

3. Call the **POM_load_instances_possible_by_enquiry** function and pass in the name of the enquiry and the class you want to load instances of. Output values are two arrays of objects, one for the objects the POM was able to load and the other for those that the POM failed to load because of Access Manager rules or a consistency check failure.

Understanding the TextServer

Whether you want to override existing messages or create your own, you need to understand Teamcenter's **TextServer**. When a client needs to display a text or error message, the **TextServer** searches a list of id-message pairs using a key id lookup, and then returns the appropriate string to the client. These id-message pairs are loaded into memory by the **TextServer** from a set of XML files.

Examples

Below are some examples of how you might use custom, localizable text messages:

- Titles for style sheets using the **titleKey** or **textKey** attributes.

- Error messages for custom code using the following **TC_API** functions:

  `EMH_store_error_s1()` through `EMH_store_error_s7()`

- Text messages with custom code using the following **TEXTSRV_API** function:

  `TXTSRV_get_substituted_text_resource()`

- Business object type and property names using the Business Modeler IDE.

> Note:
>
> Deploying a Business Modeler IDE package automatically updates the **TextServer** XML files.

TextServer locale directories

There are two main locations for these XML files.

- **TC_ROOT\lang\textserver**

  - This directory and its contents are created and maintained by the Teamcenter environment.
  - All OOTB text messages are contained in this location. It is not recommended to make any changes to these files.

- **TC_USER_MSG_DIR**

  - You must create a directory to contain your custom XML files. You must modify the **tc_profilevars** script to add this environment variable and point to your new directory.
  - Any text message resource contained in this location which duplicates an OOTB **ID** overrides the OOTB message.

Locale directory contents

The **TextServer** looks at each of these locations expecting a certain directory structure. See the structure of the OOTB location for guidance in creating your own. There are two main directory types:

Localized resources	These directories are named with Java-standard locale IDs. The contents of the current locale directory are used for translation of resources.
Unlocalized resources	This directory is named **no_translation** and is used regardless of current locale.

Add or change a resource

After creating your custom **TC_USER_MSG_DIR** location and appropriate subdirectories, Copy the following sample XML files from their OOTB location, depending on the type of message you will be creating. These files contain additional information about creating the id-message pairs.

sample_error	A starter file for creating or overriding custom error messages. Copy this file into your localized resource directories for each language.
sample_text	A starter file for nonlocalized text messages. Copy this file into your **no_translation** directory.
sample_text_locale	A starter file for localizable text messages. Copy this file into your localized resource directories for each language.

After copying the files, change the file name to something more descriptive. You can have as many of these files as you want, and their names are not important, except as noted below. The **TextServer** automatically loads all XML files in these locations, so no registration is required. It is recommended that you retain the naming convention of the files; change the **sample** prefix of the name only. All files are treated as text message resources unless the file name ends with **_errors.xml**, in which case it is treated as an error message resource.

> Note:
>
> Each time the **tcserver** process is started, the **TextServer** automatically loads the XML files, picking up any changes made. There is no need to regenerate a cache; however, a process detects changes made only when it starts, not while it is running.

ITK customization for Teamcenter applications

Cacheless Search

Creating custom logic to filter cacheless search results

The cacheless search engine provides the ability to refine the search results through customization. This capability is available by creating an extension. By extending the **BaseAction** for the **BMF_SEARCHCURSOR_process_search_results** operation of the **SearchCursor** business object, it is possible to recieve the initial search results, filter them as needed, and then returning your custom results in their place.

Procedure

Using the Business Modeler IDE:

1. Create a new **Extension** for your custom code, available for:

   Business Object: **SearchCursor**
   Operation Name: **BMF_SEARCHCURSOR_process_search_results**
   Extension Point: **BaseAction**

2. Write your custom extension code to filter the cacheless search results.

3. Add your new **Extension** to the **SearchCursor** business object as a **BaseAction**.

4. Build, package, and deploy the template.

Your custom extension code will now be executed when a cacheless search is performed.

Sample extension code snippet

Use the following code snippet as an example of how to process the search results.

```
int T9SrchCursorUserExit( METHOD_message_t* /*msg*/, va_list args )
{
    int ifail = ITK_ok;
    int num_input_search_results = 0;
    tag_t* input_search_results;

    va_list largs;
    va_copy( largs, args );
    num_input_search_results        = va_arg( largs, int );
    input_search_results            = va_arg( largs, tag_t* );
    int* num_output_search_results  = va_arg( largs, int* );
    tag_t** output_search_results   = va_arg( largs, tag_t** );
    va_end( largs );

    //Needs change - hardcode the output values to some predefined value.
    *num_output_search_results = 1;
    *output_search_results=( tag_t * )MEM_alloc( sizeof( tag_t ) * ( 1 ) );
    ( *output_search_results )[0] = input_search_results[0];

    return ifail;
}
```

Classification

Creating custom logic for autocomputing Classification attribute values

Using Classification you can autocompute values for attributes that are marked as **Autocomputed**.

The custom logic for autocomputing is divided into three parts:

- **include** statements

- Main program

  This program is used when creating the Business Modeler IDE extension definition so that it becomes the entry point for the computation logic. This program registers all the attribute dependencies and computing functions.

- Computing functions

  These functions contain the logic for manipulating attributes. Each function is registered against an autocomputed attribute, and once registered, is called directly from Teamcenter during normal execution.

1. Register autocomputed attributes.

    Each autocomputed attribute must be registered. This registration is a one-time process and needs to happen during the initialization. For each autocomputed attribute, the registration data consists of two parts:

    - Contributing dependency attributes

- Computing function

For registering attributes, use the following API:

```
ICS_auto_compute_register_attrs (
const char*          class-id,
const char*          view-id,
const int            num-attrs,
ICS_auto_compute_attr_t*     auto-compute-attrs,
logical              override-flag
)
```

Following are the values in the example.

Value	Description
class-id	Specifies the class ID for the attribute.
view-id	Specifies the view ID for the attribute. (If a view exists, it may be left blank.)
num-attrs	Specifies the number of attributes on which this attribute depends.
auto-compute-attrs	Specifies the dependency structure containing the dependency attribute IDs and a pointer to the compute function. (The structure is explained below.)
override-flag	Specifies the flag to indicate overriding of the parent class registration.

An attribute dependency structure is a new data structure specifically developed for the autocomputation of attributes:

```
typedef struct ICS_auto_compute_attr_s
{
    int auto_compute_attr_id;
    ICS_auto_compute_function_t auto_compute_function;
    int num_attr_deps;
    int attr_deps[ICS_MAX_CLASS_ATTR_SIZE];
}ICS_auto_compute_attr_t;
```

2. Implement computing functions.

For each autocomputed attribute, register some computing function. Typically, perform the following steps using the provided APIs:

a.   Read the values and properties of attributes.

b.   Take the result and execute custom attribute manipulation logic.

c.   Use the manipulated values and properties for the autocomputed attributes.

To allow this interaction with the attributes, use the following ITK APIs:

- **ICS_auto_compute_get_attr_value**

    This API retrieves the values for the given attribute.

Parameters	Description
attributeId	Specifies the attribute ID to retrieve value for.
valueCount	Specifies the number of values for this attribute.
values	Specifies the array of values for this attributes.
formats	Specifies the array of formats for values of this attribute.
unit	Specifies the array of units for values of this attribute.

- **ICS_auto_compute_set_attr_value**

    This API sets the values for the given attribute.

Parameters	Description
attributeId	Specifies the attribute ID to set value for.
valueCount	Specifies the number of values for this attribute.
values	Specifies the values for this attributes.
unit	Specifies the units for values of this attribute.

- **ICS_auto_compute_get_attr_prop**

    This API retrieves a specific property for the given attribute.

Parameters	Description
attributeId	Specifies the attribute ID to retrieve property for.
property_name	Specifies the property name to retrieve.
values	Specifies the property value.

- **ICS_auto_compute_set_attr_prop**

  This API sets a specific property for the given attribute.

Parameters	Description
attributeId	Specifies the attribute ID to set property for.
property_name	Specifies the property name to set.
values	Specifies the property value.

  If you install sample files using the Teamcenter Environment Manager (TEM), a sample PLM XML file is available for import in the *TC_ROOT***sample\in-CLASS\auto_compute** directory of your installation. This file provides an example of creating custom logic for a Classification attribute.

  A detailed description of APIs is available in the *Integration Toolkit Function Reference*.

  > Note:
  >
  > The *Integration Toolkit Function Reference* is available on Support Center.

File Management System (FMS)

Integrating a data loss prevention solution with Teamcenter

Teamcenter manages company proprietary information that is represented in metadata and in files. Teamcenter provides protection of the metadata and access to the related files. However, the files that are downloaded to the user's client machine are not protected because they are no longer in Teamcenter. Data loss prevention (DLP) solutions can control the flow of sensitive information from the client machines. For example, DLP can control how the files are emailed, burned to a CD or DVD, encrypted on external device, or uploaded to the Internet. Digital Guardian is an example of a data loss prevention solution.

You identify the desired protection for the files managed in Teamcenter. When the file transfer is completed, the DLP application is called to protect the file on the client machine. The DLP client agent enforces the specified protection. A typical example of a classified document is a CAD drawing; depending on the classification, this document may never be sent to a customer or may be sent to an internal manufacturer.

In this implementation, you define a custom character attribute on dataset objects that represents a classification code. If this attribute is set to any value (or any particular value of choice), then it indicates a dataset that needs to be protected using DLP software. Setting the value of this attribute typically happens when the dataset is created, but the mechanics of this are up to individual customizations. The custom code that hooks into the user exit (**BMF_IMF_dlp_classification_code**) needs to check whether the dataset associated with the file has this classification code set, and if so, takes appropriate action. The **BMF_IMF_dlp_classification_code** user exit is called for all file downloads.

> **Note:**
>
> The following procedure describes how to customize your environment to enable and use DLP. This procedure does not describe how to set up any specific DLP software, but rather describes how to set up the environment so that DLP hooks are enabled. In addition to performing the following steps, you must write code specific to your DLP solution, compile it, and deploy it to invoke the actual DLP software APIs.

1. In a custom template, add a property to the **Dataset** business object or any subtype:

Name	**d4ClassCode**
	The prefix depends on the template. You can name the property anything you want. This name is provided to match the sample code.
Display name	**Classification Code**
Attribute type	**Character**
Modifiable property constant setting	**Write**

2. Use one of the following methods to implement your extension:

   - User exit

     Configure the **BMF_IMF_dlp_classification_code** user exit, which is available in the Business Modeler IDE under **User Exits→File**.

     Following is the sample code for this approach:

     ```
     //@<COPYRIGHT>@
     //==================================================
     //Copyright $2014.
     //Siemens Product Lifecycle Management Software Inc.
     //All Rights Reserved.
     //==================================================
     //@<COPYRIGHT>@

     /*
      * @file
     ```

```
 *
 *    This file contains the implementation for the Extension D4dlpclscode
 *
 */

#include <D4dlpcustom/D4dlpclscode.h>

#include <ai/sample_err.h>
#include <itk/mem.h>

#include <stdio.h>
#include <stdarg.h>
#include <ug_va_copy.h>
#include <server_exits/user_server_exits.h>
#include <user_exits/user_exits.h>

#include <pom/enq/enq.h>

static int find_datatset_for_file( tag_t file_tag, tag_t *dataset_tag, char
*class_code );

int D4dlpclscode( METHOD_message_t *msg, va_list args )
{
    int ifail = ITK_ok;
    va_list largs;
    tag_t file_tag = NULLTAG;
    char* file_classification_code = 0;
    tag_t dataset_tag = NULLTAG;
    char class_code = ' ';

    msg;

    va_copy( largs, args );
    file_tag = va_arg( largs, tag_t );
    file_classification_code = va_arg( largs, char* );
    va_end( largs );

    if( ( ifail = find_datatset_for_file( file_tag, &dataset_tag, &class_code ) )
==  ITK_ok )
    {
        *file_classification_code = class_code;
    }
    else
    {
        *file_classification_code = ' ';
    }

    printf( "Invoked BMF User Exit. Classification code is: %c\n",
*file_classification_code );

    return ITK_ok;
}

int find_datatset_for_file( tag_t file_tag, tag_t *dataset_tag, char *class_code )
{
    int ifail=ITK_ok;

    static tag_t initialised = NULLTAG;
    const char *qry_name="sa::find_datatset_for_file";
```

```
            int n_cols=0,n_rows=0;
            void *** values = 0;

            *dataset_tag = NULLTAG;
            *class_code = ' ';

            if ( initialised == NULLTAG )
            {
                const char * select_attr_list[2] = {"puid", "d4ClassCode" };

                CALL( POM_enquiry_create ( qry_name ) );
                CALL( POM_enquiry_add_select_attrs ( qry_name, "Dataset", 2,
        select_attr_list ) );
                CALL( POM_enquiry_set_tag_value ( qry_name, "filetag", 1, &file_tag,
        POM_enquiry_bind_value ) );
                CALL( POM_enquiry_set_attr_expr ( qry_name, "expr1", "Dataset",
        "ref_list", POM_enquiry_in,
        "filetag" ) );
                CALL( POM_enquiry_set_where_expr ( qry_name, "expr1" ) );

                POM_cache_for_session (&initialised);
                initialised = 1;
            }
            else
            {
                CALL( POM_enquiry_set_tag_value ( qry_name, "filetag", 1, &file_tag,
        POM_enquiry_bind_value ) );
            }

            CALL( POM_enquiry_execute ( qry_name, &n_rows, &n_cols, &values ) );

            if ( n_rows >= 1 )
            {
                if( values[0][0] )
                {
                    *dataset_tag = *((tag_t *)values[0][0]);
                }
                if( values[0][1] )
                {
                    //*class_code = *((char )values[0][1]);
                    *class_code = ( (*(char *) values[0][1]));
                }
            }
            MEM_free ( values );

            return ifail;
        }
```

- Custom exit

  Only use this approach when you cannot use the Business Modeler IDE to customize the user exit.

  Following is sample code for this approach:

  ```
  //@<COPYRIGHT>@
  //==================================================
  //Copyright $2014.
  ```

```
//Siemens Product Lifecycle Management Software Inc.
//All Rights Reserved.
//================================================
//@<COPYRIGHT>@

/*
 * @file
 *
 *   This file contains the implementation for the Extension D4dlpclscode
 *
 */

#include <D4dlpcustom/D4dlpclscode.h>

#include <ai/sample_err.h>
#include <itk/mem.h>

#include <stdio.h>
#include <stdarg.h>
#include <ug_va_copy.h>
#include <server_exits/user_server_exits.h>
#include <user_exits/user_exits.h>
#include <tccore/custom.h>

#include <pom/enq/enq.h>

static int find_datatset_for_file( tag_t file_tag, tag_t *dataset_tag, char
*class_code );

int libD4dlpcustom_register_callbacks()
{
    printf( " libD4dlpcustom_register_callbacks() \n" );
    return CUSTOM_register_exit( "libD4dlpcustom", "IMF_dlp_classification_code",
(CUSTOM_EXIT_ftn_t) dl_add_class_code );
}

int dl_add_class_code( int *decision, va_list args )
{
    int ifail = ITK_ok;
    va_list largs;
    tag_t file_tag = NULLTAG;
    char* file_classification_code = 0;
    tag_t dataset_tag = NULLTAG;
    char class_code = ' ';

    va_copy( largs, args );
    file_tag = va_arg( largs, tag_t );
    file_classification_code = va_arg( largs, char* );
    va_end( largs );

    *decision = ALL_CUSTOMIZATIONS;

    if( ( ifail = find_datatset_for_file( file_tag, &dataset_tag, &class_code ) )
==  ITK_ok )
    {
        *file_classification_code = class_code;
    }
    else
    {
        *file_classification_code = ' ';
```

```
      }

      printf( "Invoked Custom User Exit. Classification code is: %c\n",
*file_classification_code );

      return ITK_ok;
}

int find_datatset_for_file( tag_t file_tag, tag_t *dataset_tag, char *class_code )
{
      int ifail=ITK_ok;

      static tag_t initialised = NULLTAG;
      const char *qry_name="sa::find_datatset_for_file";

      int n_cols=0,n_rows=0;
      void *** values = 0;

      *dataset_tag = NULLTAG;
      *class_code = ' ';

      if ( initialised == NULLTAG )
      {
            const char * select_attr_list[2] = {"puid", "d4ClassCode" };

            CALL( POM_enquiry_create ( qry_name ) );
            CALL( POM_enquiry_add_select_attrs ( qry_name, "Dataset", 2,
select_attr_list ) );
            CALL( POM_enquiry_set_tag_value ( qry_name, "filetag", 1, &file_tag,
POM_enquiry_bind_value ) );
            CALL( POM_enquiry_set_attr_expr ( qry_name, "expr1", "Dataset",
"ref_list", POM_enquiry_in,
"filetag" ) );
            CALL( POM_enquiry_set_where_expr ( qry_name, "expr1" ) );

            POM_cache_for_session (&initialised);
            initialised = 1;
      }
      else
      {
            CALL( POM_enquiry_set_tag_value ( qry_name, "filetag", 1, &file_tag,
POM_enquiry_bind_value ) );
      }

      CALL( POM_enquiry_execute ( qry_name, &n_rows, &n_cols, &values ) );

      if ( n_rows >= 1 )
      {
            if( values[0][0] )
            {
                  *dataset_tag = *((tag_t *)values[0][0]);
            }
            if( values[0][1] )
            {
                  //*class_code = *((char )values[0][1]);
                  *class_code = ( (*(char *) values[0][1]));
            }
      }
      MEM_free ( values );
```

```
        return ifail;
    }
```

3. Deploy the template changes and the binaries to the appropriate locations.

Teamcenter Mechatronics Process Management

Mechatronics Process Management fundamental objects

Teamcenter provides objects to work with functional representations, logical representations and electrical representations of electromechanical products.

Component	Objects
Electrical components	**Functionality**
	FunctionalityRevision
Signals	**Signal**
	ProcessVariable
Electrical interfaces	**GDE**
	GDEOccurrence
	Network_Port
	Connection_Terminal
Electrical connections	**PSConnection**
	Network
	Connection
	GDELink
Routes	**RouteNode**
	RouteSegment
	RouteCurve
	RoutePath
	RouteLocation
	RouteLocationRev
Allocations	**Allocation**
	AllocationMap

The following relationships allow you to associate fundamental objects:

- **Implemented By**

- **Realized By**

- **Connected To**

- **Routed By**

- **Device To Connector**

- **Assigned Location**

- **Associated System**

- **Redundant Signal**

- **Process Variable**

To control the behavior of objects and actions on those objects, you can set the following preferences:
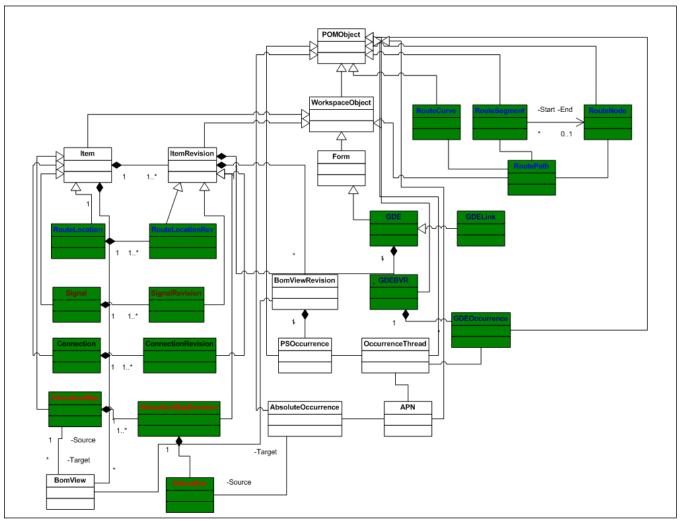
- **Connected_ToRules**

- **Implemented_ByRules**

- **Realized_ByRules**

- **APN_absolute_path_name_separator**

- **GDEOcc_display_instance_num_for_types**

- **TC_releasable_logical_types**

- **HRN_node_referenced_component_relation_secondary**

- **HRN_associated_part_relation_secondary**

Keep these preferences in mind when developing your ITK code.

Object model

The Mechatronics Process Management object model provides capabilities for representing all aspects of an electromechanical product. In the context of wire harness modeling, Mechatronics Process Management specifically provides objects to work with the functional representation, logical representation, and electrical representation of electromechanical products. Modeling an

electromechanical product in Teamcenter involves working with the objects highlighted in the Teamcenter Mechatronics Process Management object model. This model is depicted in the following figure.

Mechatronics Process Management object model

Harness model support

The classes implemented to support wire harness data include routing and topology related classes such as **Route**, **Node**, **Segment**, and **b_curve** to capture the path information associated with connections and devices implementing the connections.

Routes, segments and nodes contain a definition of the 2D or 3D routing topology associated with connections, devices or other domain specific physical objects such as wires. A route is defined as a course taken from a starting point to a destination. A route defines the course either by using one or more segments or a set of nodes in combination with curve objects, which define the shape of the route.

All route objects, such as segments, nodes, and curves, do not have any meaning outside the context of the top level assembly (in other words, a BOM view revision).

Mechatronics Process Management API modules

There are five ITK modules that you can use to customize objects to fit your needs:

- **PSCONN**

  This module provides functions that allow you to manipulate connectivity data. The functions are defined in the **psconnection.h** header file.

- **GDE**

  This module provides functions that allow you to manipulate item elements. The functions are defined in the **gde.h** header file.

- **SIGNAL**

  This module provides functions that allow you to manipulate signals. The functions are defined in the **pssignal.h** header file.

- **ROUTE**

  This module provides functions that allow you to manipulate routes. The functions are defined in the **route.h** header file.

- **ALLOC**

  This module provides functions that allow you to manipulate allocations. The functions are defined in the **allocation.h** header file.

There are example using some of the functions in the modules in the *Orientation to Mechatronics Process Management API use examples* section. You can also find sample programs using these modules in the **\samples\mechatronics** directory. Details about the module's functions can be found in the *Integration Toolkit Function Reference*.
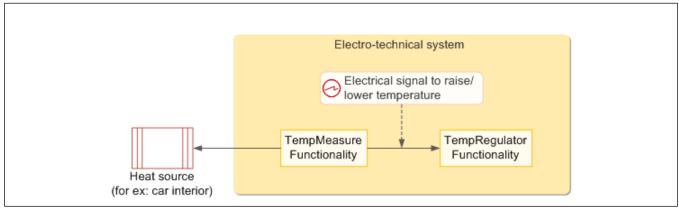
> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

You can use PLM XML to export and import data used by the Mechatronics Process Management API modules. You can also use the modules in a Multi-Site Collaboration environment.

Mechatronics Process Management API use examples

Orientation to Mechatronics Process Management API use examples

Consider the climate control system illustrated in the following figure. This system can be used in several contexts, such as an automobile or home. The system and its contents have particular meaning when used in specific contexts. Similarly, all Mechatronics Process Management objects have real meaning when used in specific contexts.

Climate control system

Creating functionality and functionality revision objects

Functionality objects are represented by a type of an item.

The following example creates a functionality object and its corresponding functionality revision object:

```
nt ifail = ITK_ok;
tag_t rev = NULLTAG;
tag_t item = NULLTAG;
ifail = ITEM_create_item( item_id,
            item_name,
            item_type,
            item_revision_id,
            &item,
            &rev );
if (ifail != ITK_ok)
return ifail;
else
{
ifail = ITEM_save_item (item);
if (ifail != ITK_ok)
  return ifail;
else
   ifail = AOM_refresh(item, false);
}
```

Creating interface port objects

An interface port must be created and added to the functionality before new ports can be defined. Interface ports are represented by a type of item element object.

The following example creates an interface port:

```
int    ifail      = ITK_ok;
tag_t  gde_obj    = NULLTAG;
ifail = GDE_create ( gde_name,
        gde_description,
        gde_type,
        &gde_obj );
if (ifail != ITK_ok)
return ifail;
else
{
ifail = AOM_save (gde_obj);
if (ifail != ITK_ok)
  return ifail;
else
  ifail = AOM_refresh(gde_obj, false);
}
```

Defining ports for a functionality object

Once interface ports are created, you can define ports for a functionality by attaching them to a revision of a functionality for a given context (view). All ports under a given item revision are tracked using an item element BOM view revision (BVR) associated with the item revision.

The following example creates the item element BVR and attaches the interface port to the functionality revision:
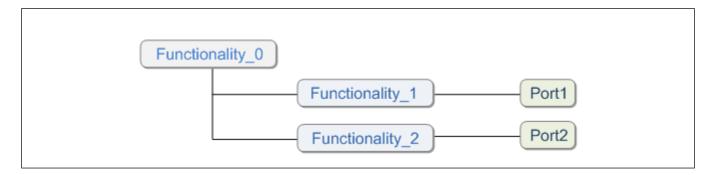
```
int    ifail      = ITK_ok;
tag_t  gde_bvr    = NULLTAG;
tag_t  gde_occ    = NULLTAG;
/* Create GDEBVR under a functionality revision */
ifail = ITEM_rev_create_gde_bvr ( rev, view_type_tag, &gde_bvr);
if (ifail != ITK_ok)
return ifail;
/* Attach the interface port to functionality revision by creating
GDEOccurrence
*/
ifail = GDE_create_occurrence ( gde_bvr,
        gde_obj,
        view_type_tag,
        occurrence_type,  /* can use NULLTAG */
        quantity,
```

```
                instance_no,
                &gde_occ);
if (ifail != ITK_ok)
return ifail;
else
{
ifail = AOM_save (gde_bvr);
if (ifail != ITK_ok)
  return ifail;
else
  ifail = AOM_refresh(gde_bvr, false);
}
```

Creating connections

Assume that you have used the APIs described above to create Functionality_0 containing Functionality_1 and Functionality_2. Functionality_1 has Port1 and Functionality_2 has Port2 (as shown in the following figure). You must connect Port1 and Port2 in a given context.

> Note:
>
> Connections are only valid for the given context. A separate connection call is required to associate a connection object with ports for each usage of a connection occurrence.

Functionality connections

The following example creates connections for Port1 and Port2:

```
int    ifail     = ITK_ok;
tag_t window = NULLTAG;
/* Create BOMWindow to display the product structure of Functionality_0 */
ifail = BOM_create_window ( &window );
if (ifail != ITK_ok)
return ifail;
ifail = BOM_set_window_pack_all ( window, TRUE );
if (ifail != ITK_ok)
return ifail;
ifail = BOM_set_window_top_line ( window,
```

```
                    functionality_1_item,
                    functionality_1_rev,
                    NULLTAG,
                    &topline );
/* Each of the components that used in the structure will be represented as
a
BOMLine. We need to get BOMLines for Port1 and Port2 so that we can use
them
in making connection.
*/
tag_t *childs  = NULL;
int count = 0;
ifail = BOM_line_ask_all_child_lines(topline, &count, &childs);
if (ifail != ITK_ok)
return ifail;
/* Look for BOMLines for Port1 and Port2 */
tag_t gdelineType = NULLTAG;
ifail = TCTYPE_find_type ("GDELine", (const char*)0, &gdelineType);
if (ifail != ITK_ok)
return ifail;
tag_t line_type_tag = NULLTAG;
tag_t line_for_port1_port2[2];
int   line_found = 0;
char* object_name = NULL;
for ( int inx =0; inx < count; inx++ )
{
 ifail = TCTYPE_ask_object_type(childs[inx], &line_type_tag);
 if (ifail != ITK_ok)
     return ifail;
 if (line_type_tag == gdelineType)
 {
    ifail = AOM_ask_value_string ( childs[inx], "bl_line_name", &obj_name );
    /* Get the BOMLine for Port1 and Port2 */
    if (! strcmp(obj_name, "Port1") || ! strcmp(obj_name, "Port2"))
    {
      line_for_port1_port2[line_found] = childs[inx];
      line_found++;
      if (line_found >= 2)
         break;
    }
    }
}
/* Create connection */
if (line_found > 0)
{
ifail = PSCONN_connect (connection_id,
          connection_name,
          connection_type,
          connection_rev,
```

```
                line_found,
                line_for_port1_port2,
                &connection_line_tag);
  }
```

Listing connections

The following example lists ports that are connected by a connection:

```
    Ifail = PSCONN_list_connected_gdes( connection_line_tag,
                    &gde_line_count,
                    &gde_line_tags);
```

Disconnecting ports from a connection

This function does not delete the connection object, but it does break the linkage between the ports and connection objects.

The following example disconnects all ports from a connection:

```
    Ifail = PSCONN_disconnect ( connection_line_tag);
```

Removing specific ports from a connection

The following example removes Port1 from the connection:

```
    tag_t  port_to_remove[*];
    int    port_count;
    port_to_remove[0] = line_for_port1;
    port_count = 1;
    ifail = PSCONN_remove_from_connection ( connection_line_tag,
                    port_count,
                    port_to_remove);
```

Creating process variables

Process variables are a type of item element. The following example creates a process variable:

```
    int    ifail      = ITK_ok;
    tag_t  process_variable_obj    = NULLTAG;
    ifail = GDE_create ( process_variable_name,
                process_variable_description,
                "ProcessVariable",   // Type of gde
                &process_variable_obj );
    if (ifail != ITK_ok)
    return ifail;
    else
```

```
{
ifail = AOM_save (process_variable_obj);
if (ifail != ITK_ok)
return ifail;
else
ifail = AOM_refresh(process_variable_obj, false);
}
```

Creating signals

Signal objects are created using the **SIG_create_signal** API as shown in the following example:

```
int ifail = ITK_ok;
tag_t rev = NULLTAG;
tag_t signal = NULLTAG;
ifail = SIG_create_signal( signal_id,
            signal _name,
            signal _type,
            signal _revision_id,
            &signal,
            &rev );
if (ifail != ITK_ok)
return ifail;
else
{
ifail = ITEM_save_item (signal);
if (ifail != ITK_ok)
return ifail;
else
ifail = AOM_refresh(signal, false);
}
```

Creating signal-process variable associations

Signals are sometimes generated due to a variation of the system variable, such as temperature or pressure. In the climate control example, the signal is generated due to a variation in car temperature compared to a preset value.

To represent this variation, a signal can be associated with a process variable of the current context using the **SIG_set_signal_pvariable** function, as shown in the following example:

```
ifail = ITK_ok;
ifail = SIG_set_signal_pvariable( signal_line_tag,
process_variable_line_tag );
if( ifail != ITK_ok )
return ifail;
```

In this example, the **signal_line_tag** and **process_variable_line_tag** tags represent the signal line and process variable line in the current context.

The **SIG_set_signal_pvariable** function requires that you set the **SIG_pvariable_rules** preference to the object type that is being passed in as secondary.

Establishing signal associated system relations

For a signal to have meaning, it must be identified with the system (associated systems) that processes the signal and its purpose. The purpose defines the role a system plays in the existence of the signal within the specified context. The standard Teamcenter installation provides the following set of roles:

- Source

- Target

- Transmitter

To set a source, target, or transmitter associated system programmatically, set the corresponding preference for that associated system to include the type of the secondary object being passed in to the function.

In the climate control example, the electrical signal is generated by the **TempMeasure** functionality and consumed by the **TempRegulator** functionality.

> Note:
>
> The association between signal and associated system is only valid in the specified context.

The following example associates the signal to the system:

```
int ifail = ITK_ok;
ifail = SIG_set_associated_system( signal_line_tag,
              asystem_line_tag,
              role,
              &relation_tag );
if( ifail != ITK_ok )
return ifail;
```

> Note:
>
> Similar to signals, process variables also have associated systems that generate, consume, or transmit the process variable. The association between process variable and associated system in a given context can be set using the **SIG_set_associated_system** function.

Removing the relation between a signal/process variable and associated system

The association between a signal or process variable and an associated system in a specified context can be removed using the **SIG_unset_associated_system** or the **SIG_unset_associated_systems** function.

You should use the **SIG_unset_associated_system** function to remove *all* associated system relations, corresponding to valid associated system roles, between a signal or process variable line tag and an associated system line tag. This function does not take multiple inputs for the signal or process variable line or the associated system line.

You should use the **SIG_unset_associated_systems** function to remove associations between a signal or process variable line tag and one or more secondary BOM lines for a *specific* role that is passed in to the function. If one or more of the secondary BOM lines passed in is invalid, or if an association does not exist between the signal or process variable line and one or more of the secondary BOM lines, that secondary line is added to a **failedLines** array output variable, and the function processes the next secondary BOM line. You must pass a valid non-null role in the **role** argument (for example, **source**, **target**, or **transmitter**).

The following example removes *all* associated system relations between a signal and associated system:

```
int ifail = ITK_ok;
ifail = SIG_unset_associated_system( signal_line_tag, asystem_line_tag );
if( ifail != ITK_ok )
return ifail;
```

In this example, the **signal_line_tag** and **asystem_line_tag** arguments represent the tags of the signal and associated system lines in a specified context.

The following example removes a *specific* associated system relation corresponding to the **source** role between a signal and secondary lines:

```
int ifail = ITK_ok;
  logical  hasFailures = false;
  int numFailedLines = 0;
  tag_t *failedLines = 0;
  ifail = SIG_unset_associated_systems( priLineTag, numSecondaries,
      secondaries, "source", &hasFailures, &numFailedLines,
&failedLines );
    if( ( ifail != ITK_ok ) || ( hasFailures ) )
    {
      // process the failures
    }
```

In this example, the **priLineTag** and **secondaries** arguments represent the tags of the primary signal or process variable line and the secondary associated system lines, respectively. The **failedLines** argument is an output array that holds the failed secondary lines. The **hasFailures** argument is a logical argument that is set to **true** if failures exist. The **numFailedLines** argument is an integer that holds the number of failed removals.

Maintaining redundant signals

To maintain proper system function, it is sometimes necessary to maintain duplicate signals in a specific context.

For example, in the **climate control example**, it is essential to maintain a redundant **TempFunctionality** functionality in the rear of the car that generates exactly the same signal as the functionality in the front of the car. This maintains proper functioning of the climate control system.

Redundant signals are identified by associating duplicate signals with the primary signal using the **SIG_set_redundant** call, as depicted in the following example:

```
int ifail = ITK_ok;
ifail = SIG_set_redundant_signal( primary_signal_line_tag,
            redundant_signal_line_tag,
            &relation_tag );
if( ifail != ITK_ok )
return ifail;
```

> Note:
>
> There can be multiple redundant signals for a given signal based on the criticality of the function being regulated.

Removing redundant signals

Redundant signals associated with a primary signal in a given context can be removed using the **SIG_unset_redundant_signal** call, as shown in the following example:

```
int ifail = ITK_ok;
ifail = SIG_unset_redundant_signal( primary_signal_line_tag,
            redundant_signal_line_tag );
if( ifail != ITK_ok )
return ifail;
```

In this example, **redundant_signal_line_tag** is the line tag of the redundant signal in the given context.

Setting signal values

Signals represent changes in the values of process variables; therefore, they carry values for the regulation of the process variables. These values can be fixed, for example 5 volts, or they can be a range, for example 5–10 volts. Because the value can vary from one revision to another, signal values are set on the signal revision objects using the **SIGREV_set_signal_value** function, as shown in the following example:

```
int ifail = ITK_ok;
ifail = SIGREV_set_signal_value( signal_revision, value );
```

```
if( ifail != ITK_ok )
return ifail;
```

In this example, **signal_revision** is the tag of the signal revision object and value is the double value to be set on the signal revision.

Setting signal characteristics

Signals represent process variables that vary in a given context; therefore, the value of the signal differs to represent the change. This variation in signal value with respect to the process variable is defined as a signal characteristic, which is represented either as an equation or as a human-readable text on each signal revision.

> Note:
>
> Teamcenter does not interpret signal characteristics on a signal revision. This is assumed to be part of the Teamcenter implementation at a given site.

The following example sets signal characteristics on a signal revision:

```
int ifail = ITK_ok;
ifail = SIGREV_set_signal_characteristics( signal_revision,
characteristics );
if( ifail != ITK_ok )
return ifail;
```

Setting signal line values

Values on signal revision objects represent a ranges or expected values that the signal is expected to generate. The actual value of a signal exists within a specific context. For example, in the **climate control example**, the value on the signal generated by the **TempMeasure** functionality depends on the exact condition (temperature) inside a car. In addition, the value generated by the **TempMeasure** functionality is different if the climate control system is used in a room or in a truck, rather than in a car.

The following example sets signal line values:

```
int ifail = ITK_ok;
ifail = SIG_set_signal_line_value( signal_line_tag, value );
if( ifail != ITK_ok )
return ifail;
```

In this example, **signal_revision** represents the tag of the signal revision object and value represents the double value to be set on the signal revision.

Limitations

- Configuration and effectivity rules are not supported by GDELine.

- There are no precise and imprecise forms of GDELine.
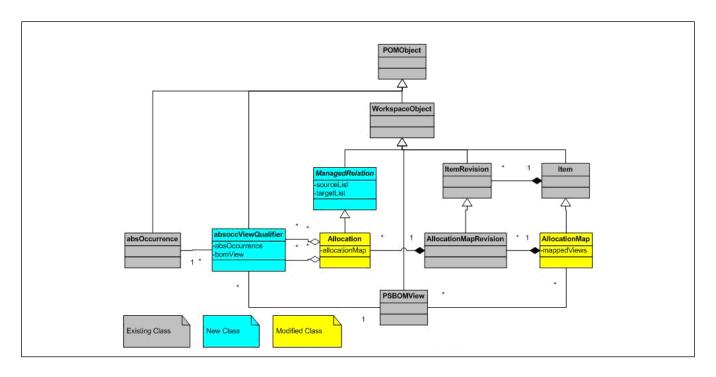
Allocations

Introduction to allocations

Allocations represent a mapping from one view in a product structure to another view. You use allocations to create separate views of a product, then map the views to each other. You can model a purely functional structure that is independent of the parts that eventually realize the functions. With careful analysis, you can use independent groups of parts to design different products with the same functional and logical models. Allocations allow you to link the actual parameters on physical parts to the requirements on the functional and logical models. This permits you to verify the components in the product are valid for their proposed purpose.
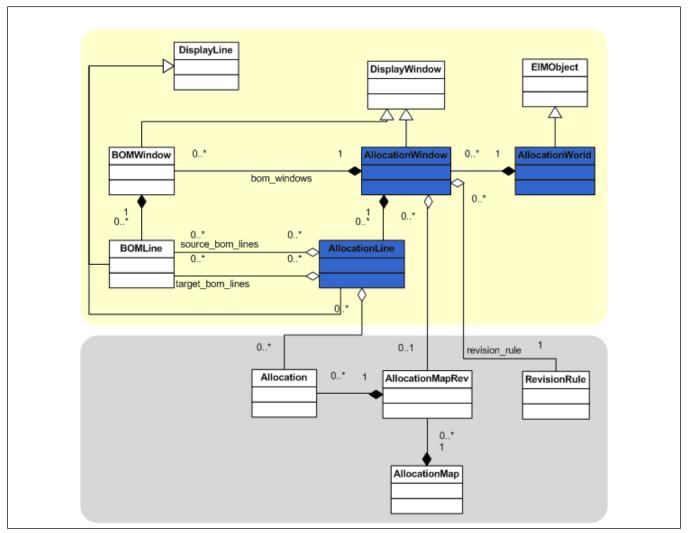
Object model

There are two allocation object models. The first, shown in the following figure, is for persistent allocations stored in the database.

Allocation persistent object model

The second model, shown in the following figure, is for run-time allocations in a window with lines that associate source BOM lines with target BOM lines.

Allocation run-time object model

Preferences

You can carry forward all allocations created in the context of an allocation map revision when it either revised, copied, or saved as a different revision by setting the **ALLOC_map_copy_allocations** preference.

You can use a preference to control the cardinality of source and targets to be associated. For example, you can enforce a **Network_Port** object to participate in one-to-one relations only with the **ALLOC_source_target_cardinality** preference.

The **ALLOC_Source_Occurrence_Types** and **ALLOC_Target_Occurrence_Types** preferences specify a list of components for a given allocation type that can participate as source and target components, respectively.

The **ALLOC_Product_Representation_Types** preference limits the view subtypes that can be used for creating allocations for a given allocation map type. The **ALLOC_Group_Allocation_Types** preference limits the type of allocations that can be used for creating allocations for a given allocation map type.

For example, you can specify that the **Allocation_One** subtype of **Allocation** be created only in association with the **AllocationMap_one** subtype of **AllocationMap**.

API functions

There are two kinds of allocation API functions: persistent and run-time. The allocation run-time model provides a simplified run-time abstraction of the allocation-related persistent object model. The run-time model includes an allocation window that contains allocation lines that associate source BOM lines with target BOM lines. Each allocation line shows attributes derived from a persistent allocation object.

Additional information about the available API functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Example

You can write a user exit to check if an allocation is complete and correct. The criteria for that are dictated by the business logic of your enterprise. Through these user exits, you can perform the validity checks from the user interface.

You can register a method against the following user exit messages:

- **ALLOC_is_allocation_complete_msg**

- **ALLOC_is_allocation_correct_msg**

For example, to register a custom method that defines the business logic against the user exit messages:

```
METHOD_register_method("AllocationLine",
ALLOCATION_is_allocation_complete_msg,
            &my_complete_allocation, NULL, &my_method))
```

This custom method returns **TRUE** if the custom conditions are satisfied, otherwise it returns **FALSE**. The conditions you implement are based on your requirements. For example, to define the correctness of an allocation if and only if the source components are of a particular type:

```
static int my_configure_allocations (METHOD_message_t *message, va_list
args)
    {
        int iFail = ITK_ok;
          // Extract the Object.
                Customer specific business logic
        //return ITK_ok;
    }
```

By default, an allocation is correct if it has a source and/or a target component.

Embedded Software Manager

Embedded software is a critical component within electronics devices. It undergoes upgrades and enhancements independent or part of electronic devices. The Embedded Software Manager (ESM) in Teamcenter manages the software components and their dependencies as well as the functional and physical components and their dependencies in an integrated product development environment. The ESM delivers four critical components to handle software management: the embedded software manager itself, embedded software explorer, the signal manager, and the signal explorer. The ITK functions corresponding to these four components are broadly divided into the **ESM_** functions and the **SIG_** functions.

The ESM is installed as an additional component during the Teamcenter installation process and requires a separate license.

Before using ITK to customize the ESM, ensure its related preferences are set to the correct software categories, types, and dataset types.

The following table lists the **ESM** module functions you can use to customize the Embedded Software Manager. These functions support the management of software, processor, and their dependencies.

Additional information about these functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Function	Definition
ESM_is_processor	Given a BOM line tag, checks whether it is a processor BOM line.
ESM_is_gateway	Given a processor BOM line tag, checks whether it is a gateway processor BOM line.
ESM_is_software	Given a BOM line tag, checks whether it is a software BOM line.
ESM_associate_processor_to_ software	Given a processor BOM line and an array of software BOM lines, associates processor and software using the **Embeds** relation. To save the associations, save the BOM window.
ESM_associate_processor_to_ processor	Given a gateway processor BOM line and an array of other processor BOM lines that are accessed through the gateway processor, associates the processors in the array with the gateway processor using the **GatewayOf** relation. To save the associations, save the BOM window.

Function	Definition
ESM_associate_software_to_ software	Given a software BOM line and an array of software BOM lines, associates software using the **Dependent On** relation. To save the associations, save the BOM window.
ESM_remove_processor_to_ software_association	Given a processor BOM line and an array of software BOM lines, removes the **Embeds** relation association of the processor and software lines. To save the associations, save the BOM window.
ESM_remove_processor_to_ processor_association	Given a gateway processor BOM line and an array of associated processor BOM lines, removes the **GatewayOf** relation association between the processor lines. To save the associations, save the BOM window.
ESM_remove_software_to_ software_association	Given a software BOM line and an array of software BOM lines, removes the **Dependent On** relation association of the software lines. To save the associations, save the BOM window.
ESM_ask_embedded_ software_of_processor	Given a processor BOM line, gets an array of software BOM lines that are associated to processor with the **Embeds** relation.
ESM_ask_gateway_of_ processor	Given a processor BOM line, gets an array of gateway processor BOM lines that are associated to it with the **GatewayOf** relation.
ESM_ask_processors_ accessedby_processor	Given a gateway processor BOM line, gets an array of processor BOM lines that are associated to it with the **GatewayOf** relation.
ESM_ask_dependent_ software_of_software	Given a primary software BOM line, gets an array of secondary software BOM lines that are dependent on the primary software.
ESM_ask_software_used_by_ software	Given a secondary software BOM line, gets an array of primary software BOM lines that are used by the secondary software.

The following table lists the **SIG** module functions you can use to customize the Embedded Software Manager. These functions support the management of signals and their dependencies.

Additional information about these functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Function	Definition
SIG_set_associated_systems	Sets the secondary lines as an associated system based on the input role to the primary signal or process variable line. This function requires you to set the appropriate preference to the correct type for the association to be created for that type. If the preference exists, but does not have a value, the corresponding association is not created.
SIG_unset_associated_ systems	Removes associations between a signal or process variable line tag and one or more secondary BOM lines for a *specific* role passed in to the function. If one or more of the secondary BOM lines passed in is invalid, or if an association does not exist between the signal or process variable line and one or more secondary BOM lines, that secondary line is added to a **failedLines** array output variable, and the function processes the next secondary BOM line. You must pass a valid non-null role in the **role** string argument.
SIG_ask_signal_sources	Finds all the sources of the signal or message line tag using the **associated_system** relation.
SIG_ask_signal_targets	Finds all the targets of the signal or message line tag using the **associated_system** relation.
SIG_ask_signal_transmitters	Finds all the transmitters of the signal or message line tag using the **associated_system** relation.
SIG_ask_device_sources	Finds all the source devices that transmit messages or signals to the target device, represented by linetag. This function uses the underlying **associated_system** relation between source and message or signal.
SIG_ask_device_targets	Finds all the targets devices that the device, represented by linetag, transmits messages or signals to. This function uses the underlying **associated_system** relation between target and message or signal.
SIG_ask_device_transmitted_ signals	Finds all the messages or signals that are transmitted by a source device, represented by linetag. This function uses the underlying **associated_system** relation between source and message or signal.
SIG_ask_device_received_ signals	Finds all the messages or signals that are received by a target device, represented by linetag. This function uses the underlying **associated_system** relation between target and message or signal.

You might want execute some of the following tasks programmatically:

- Associate a processor with the software to be installed on it.

- Associate processors with a gateway processor.

- Associate software with other software dependent on it.

The following sample is a code snippet that does the following:

- Takes a set of input BOM lines.

- Separates the gateway processor from the other processors.

- Creates the gateway relationship.

- Queries the relationship through the **ESM_ask_** functions.

- Gets the processors that are accessed through this gateway processor.

- Removes the gateway associations.

> Note:
>
> All other create, ask, and removal methods follow a similar coding pattern.

```
  // Get the gateway and the processor lines from the list of lines
 passed in
        tag_t gateway_processor_line = NULLTAG;
        for( jnx = 0 ; jnx < numBOMLines; inx++ )
        {
                If ( ESM_is_gateway( inputbomlines[inx] )
                    gateway_processor_line = inputbomlines[inx];
                else
                  {
                     If ( ESM_is_processor( inputbomlines[inx] )
                     {
                         processorLines[num_processor_lines] =
  inputbomlines[inx[;
                         num_processor_lines++;
                     }
                  }
        }
         /// associate the gateway and the processor lines
        int stat = ESM_associate_processor_to_processor(
                gateway_processor_line, num_processor_lines,
                processorLines, &hasFailures, &numFailedLines,
   &failedLines);
        for( inx = 0; inx < numFailedLines; inx++)
        {
             //Perform any custom error reporting
```

```
        }
        ….
         // next, we query for all processors accessed through this
gateway-
        // in this example, if numFailedLines was 0, the lines in
        // "processorLines" above, will match the lines in
        // accessed_processor_lines. Note that the array is not a sorted
list
        stat = ESM_ask_processors_accessedby_processor(
                gateway_processor_line, &num_accessed_lines,
                &accessed_processor_lines);
        // Now we remove the associations between the processors and
gateway
        stat = ESM_remove_processor_to_processor_association(
                gateway_processor_line, num_processor_lines,
processorLines,
                &hasFailures, &numFailedLines, &failedLines);
```

Product structure management

Bill of Materials (BOM) module

Introduction to the Bill of Materials (BOM) module

The BOM module operates at the highest level; below the Structure Manager user interface and above the PS module. It is oriented toward the presentation of and interaction with product structure and encompasses the application of configuration rules.

The BOM module is used for general operations, such as creating reports and editing structure. It pulls together information from items, item revisions, BOM view revisions and occurrences to present them as entries in a bill of materials.

The BOM module is intended to give a consistent interface to the separate elements that make up a line in a bill of materials. The design assumes that you have a BOM window (which may be a printed report or displayed in a window) showing a bill of materials made up of BOM lines. Each BOM line shows attributes derived from items, item revision, occurrences and so on; along with some attributes belonging to the BOM line itself. None of the individual objects the line represents (for example, **is_packed**) are shown.

The names of the attributes give a clue as to which object they are derived from; you can ignore that and treat them all as BOM line attributes. Although some attributes are standard and listed in the **bom_attr.h** file, others are defined at run time (for example, the note types on occurrences) and have to be found by inquiry.

The BOM module is described in two parts. The first part is for those who want to produce a report listing a bill of materials. Following this is more detail describing other functions needed for editing the bill and using more complicated facilities.

Producing a report

When producing a report, you must follow this procedure:

1. Create a BOM window.

    The first thing you have to do is create a BOM window with a call to the **BOM_create_window** function. The window may need some default settings. For example, default lines are not packed and the configuration rule is the user's default one (if defined); otherwise it is the latest with any status.

    > Note:
    >
    > To change the configuration rule, you have to ask for the configuration rule and then use the CFM ITK routines to modify it.

2. Define what to display.

    Call the **BOM_set_window_top_line** function to define what the window displays. This routine takes an item, item revision, and BOM view as arguments, but most of these can be **NULLTAG** tags.

    If the item revision is non-null, that particular revision is used. If it is **NULLTAG**, a non-null item must be given and the configuration rule is used to determine the revision.

    If the BOM view is non-null, it is used. Otherwise, the oldest one is used.

    > Note:
    >
    > Until multiple views are used, there is never more than one BOM view in an item.

3. Find children for a BOM line.

    Given a BOM line, you can find its children using the **BOM_line_ask_child_lines** function. To find attributes of a line, you have to call the **BOM_line_ask_attribute_***xxx* function, where *xxx* depends on whether you are asking about a tag, string, integer, logical or double. Before you can call the **ask_attribute** function, however, you need to find the attribute ID for the attribute you want to ask about.

4. Find an attribute ID.

    If you know what attribute you want, you can use the **BOM_line_look_up_attribute** function to translate its name to an ID. Otherwise, you have to use the **BOM_line_list_attributes** function to find all available ones and then work out which you want.

    Attributes themselves have attributes of their own. They have a name, user name, mode, read-only flag and an internal/external flag. The name is unique and is what you use to look up an attribute.

The user name is not necessarily unique and is intended as a suitable name to describe the attribute. The mode is a value defined in the **bom_attr.h** file and states if the attribute is a string, int, or some other kind.

The read-only flag is true if the attribute cannot be changed. For example, assume that the name attribute is built up from the item ID, revision ID and so on. As such, the name cannot be changed. To change the name attribute, you have to change the item ID directly.

> Note:
>
> Because an attribute is not read-only does not mean you can set that attribute on any particular line. If the line shows released data, the **set_attribute** fails.

The internal/external flag is intended to say whether the attribute was defined internally (and might be expected to be available the next time you run the program) or externally (by virtue of reflecting an occurrence note type; so it may not exist in some other run of the program).

If you are looking for data that is not available as an attribute (for example, a dataset in the item revision of some BOM line), you can use the tag attributes (item revision tag, in this case) to find the underlying object. Next, use ITK queries on the object

> Note:
>
> A simple program displaying all attributes should ignore tag attributes as being meaningless to display.

Occurrence sequencing

The single-level structure of an item revision is made up of a collection of links from a BOM view revision (the parent) to the items which are components of the assembly (the children). These links are known as occurrences.

An occurrence represents the usage of an item or an item revision within the product structure of a parent. You can distinguish between different kinds of occurrences in the BOM by referring to occurrence sequencing.

Detailed information about the BOM and product structure (PS) functions is available in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Editing the BOM

If you want to edit the bill of materials you can use the **BOM_line_cut**, **BOM_line_add** and **BOM_line_replace** functions. Be aware that the latter two use the same **NULLTAG** argument options as the **BOM_set_window_top_line** function.

If a BOM window is set to display pending edits and you made specific pending edits to a BOM line, you can remove those edits from the selected BOM line with the **BOM_line_revert_pending_edits** function or all of those pending edits in the BOM window with the **BOM_window_revert_all_pending_edits** function.

The **BOM_line_set_precise** function implements the option precise user interface command. You can use the **BOM_line_set_attribute_***xxx* functions to change those attributes that are not read-only.

If you have edited structure by direct PS ITK calls, you can re-synchronize the BOM world by calling the **BOM_update_pse** function. When you have finished your edits you must call the **BOM_save_window** function to save any changes made in that bill of materials.

> Note:
>
> The **BOM_save_window** function calls underlying PS routines to save whatever data the BOM module knows needs saving. It is possible to mix BOM and PS calls within the same program, but you must ensure that you call PS to save any change you make that the BOM does not know of.

The other functions do not affect the stored data; they only affect the display (in this case, the results of the **BOM_line_ask_child_lines** function).

Restructuring the BOM

You can add a level with the **BOM_line_insert_level** function. The levels below the new level keep their associated data. You can remove a level with the **BOM_line_remove_level** function. If you remove a level, options associated with the removed level are moved up to the level above. Variant conditions associated with the removed level are merged into their children. If you want to split a BOM line, use the **BOM_line_split_occurrence** function. The new level initially copies all associated data from the original level, including notes and variant conditions. The total quantity after the split equals the original quantity. The **BOM_line_move_to** and **BOM_line_copy** functions move and copy the BOM line to a new place in the structure, respectively. Absolute occurrences are preserved during all restructuring operations.

Using global alternates

A global alternate part is interchangeable with another part in all circumstances, regardless of where the other part is used in the product structure. A global alternate applies to any revision of the part and is independent of any views. Parts and their global alternates are related only in a single direction. Use the following functions to manage global alternates of a single item, such as listing, adding, removing, and setting the preferred global alternate:

- **ITEM_list_related_global_alternates**

  Lists related global alternates.

- **ITEM_add_related_global_alternates**

  Adds global alternates.

- **ITEM_remove_related_global_alternates**

  Removes global alternates.

- **ITEM_prefer_global_alternate**

  Makes the specified global alternate the preferred global alternate for the specified item.

- **ITEM_ask_has_global_alternates**

  Checks if the specified item has any global alternates.

Packing the BOM

Packing is controlled at the window level by a default flag. If you call the **BOM_line_pack** or **BOM_line_unpack** function on a given line, however, the results of later calls to **ask_child_** lines on its parent change. By default, lines are packable if they have the same item revisions and the same find numbers. By calling the **BOM_set_pack_compare** function, you can supply a different compare function. This function applies session-wide; it is not window-specific.

> Note:
>
> The interactive Structure Manager assumes that all packed lines show the same find number. If the user changes the find number, it sets all of the lines to that number. If you change this function, Structure Manager might not work correctly.

Sort functions

The **BOM_line_ask_child_lines** function lists the lines in some order determined by a sort function. The default function sorts by find number and then by item name. The **BOM_set_window_sort_compare_fn** function can be used to change this function on a per window basis.

Sample BOM program

The following code shows how the BOM ITK functions can be used:

```
/* An example program to show a list of a bill of materials using the BOM module
@<DEL>*/
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unidefs.h>
#include <itk/mem.h>
#include <tc/tc.h>
#include <tccore/item.h>
#include <bom/bom.h>
#include <cfm/cfm.h>
#include <ps/ps_errors.h>

/* this sequence is very common */
#define CHECK_FAIL if (ifail != 0) { printf ("line %d (ifail %d)\n", __LINE__, ifail);
    exit (0);}
static int name_attribute, seqno_attribute, parent_attribute, item_tag_attribute;
static void initialise (void);
static void initialise_attribute (char *name,  int *attribute);
static void print_bom (tag_t line, int depth);
static void print_average (tag_t top_line);
static void find_revision_count (tag_t line, int *count, int *total);
static void double_find_nos (tag_t line);
static int my_compare_function (tag_t line_1, tag_t line_2, void *client_data);
/*-------------------------------------------------------------------------*/
/* This program lists the bill under some given item (configuring it by latest+working).
   It then doubles all the find numbers and re-lists it by descending find number
   order. It then counts how many lines there are in the bill and the average number of
   revisions of the items in that bill.
   It never actually modifies the database.
   Run this program by:
      <program name> -u<user> -p<password>
*/
extern int ITK_user_main (int argc, char ** argv )
  {
    int ifail;
    char *req_item;
    tag_t window, window2, rule, item_tag, top_line;
    initialise();
    req_item = ITK_ask_cli_argument("-i=");
    ifail = BOM_create_window (&window);
    CHECK_FAIL;
    ifail = CFM_find( "Latest Working", &rule );
    CHECK_FAIL;
    ifail = BOM_set_window_config_rule( window, rule );
    CHECK_FAIL;
    ifail = BOM_set_window_pack_all (window, true);
    CHECK_FAIL;
    ifail = ITEM_find_item (req_item, &item_tag);
    CHECK_FAIL;
    if (item_tag == null_tag)
      { printf ("ITEM_find_item returns success,  but did not find %s\n", req_item);
        exit (0);
      }
    ifail = BOM_set_window_top_line (window, item_tag, null_tag, null_tag, &top_line);
    CHECK_FAIL;
    print_bom (top_line, 0);
    double_find_nos (top_line);
    /* we will not use
        ifail = BOM_save_window (window);
      because that would modify the database
    */
```

```
      /* now let's list the results sorted the other way */
      ifail = BOM_create_window (&window2);
      CHECK_FAIL;
      ifail = BOM_set_window_config_rule( window2, rule );
      CHECK_FAIL;
      ifail = BOM_set_window_pack_all (window, false);
         /* the default, but let's be explicit */
      CHECK_FAIL;
      ifail = BOM_set_window_sort_compare_fn (window2, (void *)my_compare_function, NULL);
      CHECK_FAIL;
      ifail = BOM_set_window_top_line (window2, item_tag, null_tag, null_tag, &top_line);
      CHECK_FAIL;
      printf ("===============================================\n");
      print_bom (top_line, 0);
      print_average (top_line);
      ITK_exit_module(true);

      return 0;
   }
/*-----------------------------------------------------------------------*/
static void print_bom (tag_t line, int depth)
   {
      int ifail;
      char *name, *find_no;
      int i, n;
      tag_t *children;
      depth ++;
      ifail = BOM_line_ask_attribute_string (line, name_attribute, &name);
      CHECK_FAIL;
      /* note that I know name is always defined, but sometimes find number is unset.
         If that happens it returns NULL, not an error.
      */
      ifail = BOM_line_ask_attribute_string (line, seqno_attribute, &find_no);
      CHECK_FAIL;
      printf ("%3d", depth);
      for (i = 0; i < depth; i++)
        printf ("  ");
      printf ("%-20s %s\n", name, find_no == NULL ? "<null>" : find_no);
      ifail = BOM_line_ask_child_lines (line, &n, &children);
      CHECK_FAIL;
      for (i = 0; i < n; i++)
        print_bom (children[i], depth);
      MEM_free (children);
      MEM_free (name);
      MEM_free (find_no);
   }
/*-----------------------------------------------------------------------*/
static void double_find_nos (tag_t line)
{
      /* just to demonstrate modifying the bill */
      int ifail;
      char *find_no;
      int i, n;
      tag_t *children;
      ifail = BOM_line_ask_attribute_string (line, seqno_attribute, &find_no);
      CHECK_FAIL;
      /* Top bom lines have no find number, and others may also not */
      if (find_no[0] != '\0')
      {
          char buffer[100];
```

```
        sprintf (buffer, "%d", 2 * atoi (find_no));
        ifail = BOM_line_set_attribute_string (line, seqno_attribute, buffer);
        if (ifail != ITK_ok)
        {
            char *child_name;
            ifail = BOM_line_ask_attribute_string (line, name_attribute, &child_name);
            CHECK_FAIL;
            printf ("==> No write access to %s\n", child_name);
            MEM_free (child_name);
        }
        else
        {
            CHECK_FAIL;
        }
        MEM_free (find_no);
    }
    ifail = BOM_line_ask_child_lines (line, &n, &children);
    CHECK_FAIL;
    for (i = 0; i < n; i++)
        double_find_nos (children[i]);
    MEM_free (children);
}
/*----------------------------------------------------------------------------*/
static void print_average (tag_t top_line)
{
    int count = 0, total = 0;
    find_revision_count (top_line, &count, &total);
    if (count <= 0)
    {
        printf ("impossible error has happened!\n");
    }
    else
    {
        printf ("lines in bill : %d\naverage revisions of each item : %d\n", count, total
         / count);
    }
}
/*----------------------------------------------------------------------------*/
static void find_revision_count (tag_t line, int *count, int *total)
{
    /* A function to demonstrate going from BOM tags to other classes          */
    /* In this case, we get the Item tag for the BOM line, and then use it for an  */
    /* ITEM call. For a complete list of standard attributes see bom_attr.h, or   */
    /* use a BOM_list_attributes call to find all current attributes (new note    */
    /* types define new attributes)                                            */
    int ifail;
    tag_t item_tag, *revisions, *children;
    int i, n, revision_count;
    char*  item_id;
    ifail = BOM_line_ask_attribute_tag(line, item_tag_attribute,  &item_tag );
    CHECK_FAIL;
    /* the simplest example call I can think of:       */
    /* count how many revisions of this Item there are */
    ifail = ITEM_list_all_revs (item_tag, &revision_count, &revisions);
    CHECK_FAIL;
    MEM_free (revisions);
    (*count)++;
    (*total) += revision_count;
    ifail = BOM_line_ask_child_lines (line, &n, &children);
    CHECK_FAIL;
```

```
      for (i = 0; i < n; i++)
          find_revision_count (children[i], count, total);
      MEM_free (children);
  }
  /*----------------------------------------------------------------------*/
  static int my_compare_function (tag_t line_1, tag_t line_2, void *client_data)
    {
      /* returns strcmp style -1/0/+1 according to whether line_1 and line_2 sort <, = or > */
      char *seq1, *seq2;
      int ifail, result;
      ifail = BOM_line_ask_attribute_string (line_1, seqno_attribute, &seq1);
      CHECK_FAIL;
      ifail = BOM_line_ask_attribute_string (line_2, seqno_attribute, &seq2);
      CHECK_FAIL;
      if (seq1 == NULL || seq2 == NULL)
        result = 0;
      else
        result = strcmp (seq2, seq1);  /* Doing a reverse sort */
      /* note:  the default sort function compares Item names if the find numbers sort
         equal but we will not show that here
      */
      MEM_free (seq1);
      MEM_free (seq2);
      return result;
    }
  /*----------------------------------------------------------------------*/
  static void initialise (void)
    {
      int ifail;
      /* <kc> exit if autologin() fail */
      if ((ifail = ITK_auto_login()) != ITK_ok)
          fprintf(stderr,"Login fail !!: Error code = %d \n\n",ifail);
      CHECK_FAIL;
      /* these tokens come from bom_attr.h */
      initialise_attribute (bomAttr_lineName, &name_attribute);
      initialise_attribute (bomAttr_occSeqNo, &seqno_attribute);
      ifail = BOM_line_look_up_attribute (bomAttr_lineParentTag, &parent_attribute);
      CHECK_FAIL;
      ifail = BOM_line_look_up_attribute (bomAttr_lineItemTag, &item_tag_attribute);
      CHECK_FAIL;
    }
  /*----------------------------------------------------------------------*/
  static void initialise_attribute (char *name,  int *attribute)
    {
      int ifail, mode;
      ifail = BOM_line_look_up_attribute (name, attribute);
      CHECK_FAIL;
      ifail = BOM_line_ask_attribute_mode (*attribute, &mode);
      CHECK_FAIL;
      if (mode != BOM_attribute_mode_string)
        { printf ("Help,  attribute %s has mode %d,  I want a string\n", name, mode);
          exit(0);
        }
    }
```

Occurrences

An occurrence represents the usage of an item or an item revision within the product structure of a parent. You can distinguish between different kinds of occurrences in the BOM and product structure.

Additional information about occurrence functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Options and variations

Options and variations basic concepts

The basic concepts behind options and variations are that options are used to customize the BOM and variant expressions that reference these options at run time.

- **Expressions**

  Expressions are built as a parse-tree of variant expressions. A collection of these expressions is held in a variant expression block. The same mechanism is used to attach variant conditions to an occurrence and attach variant declarations to an item revision. The restriction is that only appropriate types of expressions may be attached to each parent. With an occurrence, the variant expression block may only contain one expression and must be a **LOAD-IF** type of expression. With an item revision, many types of expressions may be used to declare an option, set a value or apply an error check.

- **Range of Allowed Values**

  Each variant expression contains an operator, a left operand and a right operand. Options are defined to belong to some item, but the range of allowed values is stored in option revisions. In most cases the bottom of an expression tree is an option, but the declare operator references an option revision to identify the range of allowed values current assembly option.

- **Restrictions**

  Options are only enumerated values. Expression trees only evaluate to logical or integer values.

If you want to print a report showing variant conditions on a BOM or ignore BOM lines that are not selected by the current option values, use ITK enquiries for BOM line values rather than trying to evaluate variant expression trees. The ITK routines documented here are very low-level routines that are intended to be used to create variant conditions when building a BOM.

Variant operator values are defined in the **bom_tokens.h** file. The **BOM_variant_op_rhs_is_string** value can be added to other operators to allow a string to be used as the right operand. In most cases the right operand should be an integer value that is stored as a text string containing decimal digits.

Operators

The following table lists each variant expression operator, the type of left and right operands allowed in that expression, and a brief description.

> Note:
>
> The operator names in the following table are truncated for convenience. The full name is the operator name in the following table prefixed with **BOM_variant_operator_**. For example, **declare** in the table is **BOM_variant_operator_declare**.

Operator	Left operand type	Right operand type	Description
declare	option revision	**NULLTAG**	If this option has not yet been declared for this BOM, then declare it as this revision.
assy_uses	option	**NULLTAG**	Not used.
default	option	expression or string	Sets the option to the specified value unless the option has already been set. The value is an integer index into the allowed range of enumerated values.
assign	option	expression or string	Sets the option to the specified value unless it has been previously set. Returns an error if the option already has a different value.
error_if	expression	string	Used to implement error checks. Reports the string as an error if expression is true.
if	do-expression	condition–expression	Evaluates do-expression if condition-expression is true. This is used to implement derived values by making the do-expression set a default value.
is_equal	expression	expression	Compares the two expressions. Typically the first expression is an option and the second a string in order to verify that the option is set to the specified value.
not_equal	expression	expression	Compares the two expressions. Typically the first expression is an option and the second a string in order to verify that the option is not set to the specified value.
and	expression	expression	Evaluates the logical **AND** of the two expressions.

Operator	Left operand type	Right operand type	Description
or	expression	expression	Evaluates the logical **OR** of the two expressions.
load_if	**NULLTAG**	expression	The top operator of the expression in a variant condition attached to an occurrence.
comment	**NULLTAG**	string	Not used.
rule_set	option	expression or string	Used by variant rule objects to store option-value pairs. This should not be used in any other context.
brackets	**NULLTAG**	expression	Placeholder for marking sub-expressions which the user would like to be bracketed. This operator has no effect on the evaluation of the overall expression, it is only used when displaying the expression in text form.

Run-time options

Once the assembly has been built to include variations data, option values can be used to determine what parts of the assembly to display at run time. Options can be listed and set from a specified window using the **BOM_window_ask_options** and **BOM_window_set_option_value** functions, respectively. If the desired option is known, but it is not known whether it has been previously referenced, use the **BOM_window_find_option** function instead of the **BOM_window_ask_options** function. The following code shows an example of listing and setting option values:

```
#include <string.h>
#include <unidefs.h>
#include <tc/tc.h>
#include <bom/bom.h>
void panic();
void error(const char* message);
void set_option(tag_t w, const char *option, const char *value)
{
    /* Sets an option to the specified value in Structure Manager window w. */
    tag_t *options, *option_revs;
    tag_t found_option, found_option_rev;
    int *indexes;
    int i, n;
    if (BOM_window_ask_options(w, &n, &options, &option_revs) != ITK_ok) panic();
    for (i = 0; i < n; i++)
    {
        char *name, *description;
        tag_t owning_item;
        logical found;
        if (BOM_ask_option_data(options[i], &owning_item, &name, &description)!= ITK_ok)
```

```
        panic();
    found = (strcmp(name, option) == 0);
    MEM_free(name);
    MEM_free(description);
    if (found) break;
}
if (i == n) error("option not used in this window");
found_option = options[i];
    found_option_rev = option_revs[i];
MEM_free(options);
MEM_free(option_revs);
if (BOM_list_option_rev_values(found_option_rev, &n, &indexes) != ITK_ok) panic();
for (i = 0; i < n; i++)
{
    char *opt_value;
    logical found;
    if (BOM_ask_option_rev_value(found_option_rev, indexes[i], &opt_value)!= ITK_ok)
    panic();
    found = (strcmp(opt_value, value) == 0);
    MEM_free(opt_value);
    if (found) break;
}
if (i == n) error("option not allowed to have that value");
/* now do the work */
if (BOM_window_set_option_value(w, found_option, indexes[i]) != ITK_ok) panic();
MEM_free(indexes);
}
```

Option creation and declaration

When creating a new option with the **BOM_create_option** function, it is necessary to declare the option revision against an item revision. You can both create and declare an option with a single call to the **BOM_new_option** function. To declare an existing option against a different item revision, call the **BOM_declare_option** function.

BOM variant rule

The BOM variant rule is a run-time object that represents the list of options and their values in a BOM window. In general, you can ignore this object and use the **BOM_window_** variant functions to get and set option values.

However, this object is for creating multiple BOM variant rules against a BOM window; although only one is actually applied to the BOM window at any one time. This can be used to set multiple option values without reconfiguring the entire BOM after each set operation, although option defaults and rule checks are still triggered. Then, once all options have been set, the BOM variant rule can be applied to the BOM window.

The BOM variant rule supports an ITK interface that is almost identical to the BOM window variant functions for getting and setting options and their values. A BOM window's current BOM variant rule can be obtained by a call to the **BOM_window_ask_variant_rule** function.

Variant expression manipulation

In an attempt to simplify Variant Expression interaction at the ITK level, some high-level concepts have been introduced. A Variant Clause List can be used to create and edit conditional variant expressions (for example, **Engine = 1.6 AND Gearbox = manual**). These conditional expressions can then be used to create IF type Variant Expressions (for derived defaults, rule checks, and occurrence configuration) using the simple **BOM_variant_expr_** ITK calls.

Variant clause list

Variant clause lists are used to store conditional variant expressions in a more easily understood (and edited) format. For example, the condition:

```
"Engine = 1.6 OR Engine = 1.8 AND Gearbox = manual"
```

would be represented in a clause list as:

```
"Engine = 1.6"
"OR Engine = 1.8"
"AND Gearbox = manual"
```

In other words, each clause is a simple condition and a join operator (**AND** or **OR**).

Clauses may be added (anywhere in the list), deleted, moved, or queried. Individual clauses within the list are identified by their position. The first clause is at position 0, the second at position 1, and so on.

Brackets are supported by variant clause lists. To add brackets around a section of a clause list, call the **BOM_variant_clause_toggle_brackets** function and pass it the array of clause positions to be contained within the brackets. For example, to place brackets around the **Engine = 1.8 AND Gearbox = manual** section of the previous example, pass in an array containing the positions 1 and 2.

> Note:
>
> To add brackets around a clause list section that contains more than two clauses, pass in the positions of the first and last clauses in the section. The other positions are ignored. The array mechanism is used, in preference to only supplying first and last position arguments, to simplify user interface implementations by allowing you to simply pass in the list of currently selected clauses, rather than expecting you to filter the selection list down to just the first and last.

Modular variants

If you work with modular variants, you must take a different approach in your ITK code. For example, if you want to create a BOM selected option set, set the option values, and then apply the set, you:

1.   Ask for the selected option set from a BOM line:

```
BOM_line_ask_sos( bl, &sos )
```

> **Note:**
>
> There are two kinds of option sets: a run-time selected option set and a persistent stored option set kept in the database.

2. Get its contents to set its options and values:

```
BOM_sos_ask_entries( sos, entry_count, options, paths )
```

3. Set the value for each option. In this call, the function sets **option1** with the value of **Economy**:

```
BOM_sos_set_entry_string( sos, options[ 0 ], "", "Economy", 4 )
```

4. Apply the set:

```
BOM_sos_apply( sos, true )
```

If you want to create a stored option set in the database and then read it, you:

1. Ask for the selected option set from the BOM line:

```
BOM_line_ask_sos ( bomline1, bomsos )
```

2. Create a variant configuration object. The first variable is given if you have a classic variant rule:

```
BOM_create_variant_config ( NULLTAG, 1, { bomsos },
bom_variant_config )
```

3. Set the option and values as well as the mode:

```
BOM_sos_set_entry_string ( bomsos, opt, "", "value1",
BOM_option_set_by_user )
BOM_sos_set_entry_logical ( bomsos, opt1, "", true,
BOM_option_set_by_user )
```

Then create a stored option set in the database from the BOM variant configuration:

```
BOM_sos_db_create("bomsos1", bom_variant_config, sos1)
```

4. Read the saved stored option set contents from the database:

```
BOM_sos_db_contents(sos1, count, items, options, optionTypes,
valueTypes,
   howSet, values)
```

If you want to read and load the contents of the stored option set saved in the database, you:

1. Ask for the selected option set from the BOM line:

    ```
    BOM_line_ask_sos ( bomline1, bomsos2 )
    ```

2. Create a variant configuration object:

    ```
    BOM_create_variant_config ( NULLTAG, 1, { bomsos2 },
    bom_variant_config2 )
    ```

3. Read the stored option set from the database:

    ```
    BOM_sos_db_read ( sos1, bom_variant_config2 )
    ```

4. Get the value of the option in the stored option set:

    ```
    BOM_sos_ask_entry_display ( bomsos2, opt, "", value, howset )
    ```

5. Apply the variant configuration:

    ```
    BOM_variant_config_apply ( bom_variant_config2 )
    ```

Exporting configured assemblies to NX

You can configure a product structure with revision rules and variants, and then export the configured structure to NX in native format. Use the **BOM_export_configured_nx_assembly** function in the BOM module.

Additional information about this function can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

BOM Compare functions

Scope of compare

BOM Compare operates on the contents of two BOM windows. This means that it picks up the configuration rules of the BOM windows from which the compared BOM lines were taken. If you want to compare different configurations of the same BOM, first set up two BOM windows with the required configurations.

By default, the **BOM_compare** function compares everything below the supplied BOM lines right down to the bottom of the BOM. It automatically loads any required structures.

If a you want to prevent a compare from loading the components of a specific structure, set the **stop** flag of the structure's BOM line, using the **BOM_line_set_stop** function. The **stop** flag can be cleared using the **BOM_line_clear_stop** function. In Structure Manager-based compares, the **stop** flags are set on all visible leaf nodes and collapsed structure nodes.

Compare descriptor

The compare descriptor is a list of attributes (referred to as compare elements to differentiate them from POM attributes) that are used as the basis of the compare. Although most compare elements are simply be properties (and are handled automatically by the compare engine), you can compare based on nonproperty elements by defining methods. You can also use methods to customize the compare behavior for awkward properties (for example, quantity, which is described in this section).

Compare elements can be generally divided into two distinct groups: primary elements and aggregate elements (there's a third group: display elements, which is described in this section). The key difference between these two groups is that primary elements define different entities, while aggregate elements are combined for multiple instances of the same entity. For example, the compare functionality before version 8 (when expressed in version 8 terms) defined item ID as a primary element and quantity as an aggregate element. This meant that BOM lines that had the same item ID were bundled together for the purposes of compare and their quantities were added together. In other words, **1 BOMLine with qty 2** was equivalent to **2 BOMLines with qty 1**.

In the following code example, the compare descriptor uses the **bl_occurrence** (occurrence thread tag) property, the **bl_item** property, the **bl_quantity** property, and the **Torque** (an occurrence note) property.

BOM Compare mode

The BOM Compare mode is a uniquely named object that combines a compare descriptor and a BOM traversal order. The BOM traversal order defines which BOM lines are fed into the compare engine, and in what order. The BOM Compare supports three traversals: single-level, multilevel, and lowest level. A single descriptor can be shared between multiple compare modes.

The following code shows a single-level compare which is best suited for an occurrence-based compare:

```
int define_compare_mode()
{
    int ifail;
    tag_t desc;
    ifail = CMP_find_desc("example desc", &desc);
    if (ifail != ITK_ok)
        return ifail;
    if (desc == NULLTAG)
    {
        ifail = define_compare_desc(&desc);
        if (ifail != ITK_ok)
            return ifail;
    }
    ifail = BOM_compare_define_mode(
        "example mode",
        BOM_compare_singlelevel, /* from bom_tokens.h */
```

```
        desc,                       /* from above */
        false,                      /* not shown UI - irrelevant*/
        false,                      /* no autopack - only multi-level needs this */
        true                        /* virtual unpack - copes with packed bomlines */
                                    /* All occurrence—based compares need this   */
            );
    if (ifail != ITK_ok)
    {
        /* If it was previously defined, return success */
        if (ifail == BOM_duplicate_bom_compare_mode)
        {
            EMH_clear_errors();
            return ITK_ok;
        }
        /* otherwise report failure */
        return ifail;
    }
    return ITK_ok;
}
int define_compare_desc(tag_t *desc)
{
    tag_t new_desc;
    tag_t new_element;
    tag_t bomline_type;
    /* Insert error handling here */
    CMP_create_desc("example desc", &new_desc);
    TCTYPE_find_type("BOMLine", NULL, &bomline_type);
    /***** Primary Element : Occurrence Thread *****/
    CMP_create_prop_element(
        new_desc,
        CMP_primary_element,
        CMP_cache_sync,      /* Cache property values in native type */
        NULL,                /* Do not define element name - will default to prop name */
        false,               /* Show value of this property in output */
        bomline_type,        /* Type to which prop belongs */
        "bl_occurrence",     /* Name of property */
        NULL,                /* No display prop - use real prop instead */
        &new_element);
    /***** Aggregate Element : Item *****/
    CMP_create_prop_element(new_desc,
                            CMP_aggregate_element,
                            CMP_cache_sync,
                            NULL,
                            false,
                            bomline_type,
                            "bl_item",
                            "bl_item_item_id",
                            &new_element);
    /***** Aggregate Element : Quantity *****/
    CMP_create_prop_element(new_desc,
                            CMP_aggregate_element,
                            compareCacheSync,
                            "k_compare_qty",
                            false,
                            bomline_type,
                            "bl_quantity",
                            NULL,
                            &new_element);
    /***** Aggregate Element : Torque Occ Note *****/
    CMP_create_prop_element(new_desc,
```

```
                                    CMP_aggregate_element,
                                    CMP_cache_sync,
                                    NULL,
                                    false,
                                    bomline_type,
                                    "Torque",
                                    NULL,
                                    &new_element);
        *desc = new_desc;
        return ITK_ok;
}
```

For simple examples, defining a compare mode is relatively easy. You create your descriptor and create a number of property elements against it. Then define your mode, combining the descriptor with a traversal mode.

There are a number of features that are described as follows:

- **UI suppression**

    In the call to the **BOM_compare_define_mode** function, the fourth argument is the mode UI display flag. This controls whether the UI should present this compare mode in the **Structure Manager BOM Compare** dialog box. This is a legacy setting and is now irrelevant because even if this was set to true, the UI still would not show this mode in the dialog box. The UI dialog box is currently hard coded to our standard pre-version 8 compare modes.

- **Auto pack**

    In the call to the **BOM_compare_define_mode** function, the fifth argument defines whether multilevel compares should pack BOM lines as it goes along. Auto pack is an attempt to allow the simplification of the presentation of the results of multilevel compares where the primary compare elements match the pack rule attributes. By default, the BOM line pack attributes are item ID and find number. These are the primary attributes for the standard multilevel compare as currently supported by the Structure Manager compare dialog box.

    If you set this argument, set it to false.

- **Virtual unpack**

    In the call to the **BOM_compare_define_mode** function, the sixth argument defines whether BOM lines should be temporarily unpacked during the compare. If you are using **bl_occurrence** as one of your compare elements, set this flag to true because packed BOM lines represent multiple occurrences. To gain access to the individual occurrences, the compare must unpack the BOM line, pull out the required data, and then repack, which is transparent to the end user. Why is not this always switched on? The main limitation of virtual pack lies in the BOM line property compare output. Normally, BOM compare sets a BOM line property called **Changes** to display the actual changes that have happened to the BOM line (for example, **Qty 2→3** or **Rev A→B**). If you are performing an occurrence-based compare, you might have two occurrences behind a packed BOM line. One occurrence might have a change (for example, **Qty 2→3**). The other occurrence might not have any change. In this situation, what should compare put in the **Changes** property of the BOMLine? Should

it say **Qty 2→3**? Should it be empty? In reality, it says **PACKED CHANGES**, and expects the user to manually unpack the BOM line to see to individual changes.

For non-UI compares, Siemens Digital Industries Software recommends that the BOM windows being compared have packing switched off.

- **Caching**

  In the calls to the **CMP_create_prop_element** function, the third argument tells BOM Compare whether and how to cache property values. The **CMP_cache_sync** setting means that the type of the cache should be synchronized to the property value type. Setting this to **CMP_cache_none** would disable caching of that property. The **CMP_cache_auto** setting is similar to **CMP_cache_sync**, but allows BOM Compare to make the final decision on the cache type. This is important if you plan to use certain properties as aggregate elements. For example, multiple tag elements cannot be aggregated into a tag cache. Instead, they need a tag array cache type. The **CMP_cache_auto** setting makes that decision for you. You can also manually specify the type of the cache, but it is up to you to make sure that it is suitable for storing the property.

- **Element naming**

  In the calls to the **CMP_create_prop_element** function, the fourth argument allows a user to specify the (internationalized) name by which this element is known. If this is set to NULL, then the property name is used. Element naming is primarily used for nonproperty compare elements, but can still be useful for property elements where the property name is considered to be too verbose for display in compare output. For example, with the **bl_quantity** property you might want compare to output the more compact **Qty** rather than property display name **Quantity**.

- **Element display suppression**

  In the calls to the **CMP_create_prop_element** function, the fifth argument allows for the possibility that certain elements are useful for comparing, but not good for displaying. For example, tag elements are great for comparing, but because they are simply unsigned integers they have no meaning to end users. This ties into display elements.

- **Element display aliasing**

  In the calls to the **CMP_create_prop_element** function, the eighth argument provides a simplified version of the display elements concept. It allows an unfriendly property to be replaced in the output stage with a more readable one. In the example above, the **bl_item** property is used. This property is the tag of the item. When displaying item changes to the user, you do not want to show them item tags. Instead you would want to show them the item ID. To do this, you could define **bl_item_item_id** as a display alias property of the **bl_item** property element. Note that while you could use **bl_item_item_id** as your compare element, the use of **bl_item** is much more efficient for two reasons: it is an integer compare rather than a string compare, and **bl_item** is a directly defined BOM line property while **bl_item_item_id** is indirected through the item.

BOM Compare execution

The following code shows the code for a simple compare report:

```
tag_t compare;
    int n_lines;
    char **lines;
    tag_t *cmp_items;
    int i;
    BOM_compare_create(&compare);
    BOM_compare_execute(compare,
                        bomline1, bomline2,
                        "example mode",
                        BOM_compare_output_report);
    BOM_compare_report(compare, &n_lines, &lines, &cmp_items);
    for (i=0; i<n_lines; i++)
    {
        printf("%s\n", lines[i]);
        /*
        ** BOMLines relating to this report line can be obtained via
        ** BOM_compare_list_bomlines(cmp_items[i], ...);
        */
    }
    AOM_delete(compare);
```

If you want the changes written to the BOM line **Changes** property, just add the **BOM_compare_output_bomline** argument to the **BOM_compare_output_report** argument in the call to the **BOM_compare_execute** function.

Display elements

In addition to primary and aggregate elements, BOM Compare can also support display elements. These are elements that have no impact on the actual comparison. Display elements are extra context data that is written as part of compare output. This is just an advanced version of the element display aliasing concept. Aliasing is limited in that it only works with property elements, and only a single alias property can be defined. Display elements can be defined with methods, allowing nonproperty data to be used. You can also define as many display properties as you want.

You can use the **bl_item_item_id** alias as a display alias for **bl_item**. If you want to display the item name as well as the item ID, define a property display element for the **bl_item_object_name** property in the descriptor, as shown in the following code:

```
CMP_create_prop_element(new_desc,
                        CMP_display_element,
                        CMP_cache_sync,
                        NULL,
                        false,
                        bomline_type,
```

```
"bl_item_object_name",
NULL,
&new_element);
```

Note that display elements are not aggregated. Teamcenter assumes that the display element property value is constant across all BOM lines in a set. As such, the display element value is simply pulled out of the first BOM line that the compare engine finds.

Aggregate element uniqueness

With numeric elements, aggregation is easy: the numbers are just added together. However, with string elements, aggregation is more difficult. The standard behavior is simply to concatenate the strings into a comma separated list, in alphabetical order. For example, **A** and **B** becomes **A,B**. If you aggregate **A** and **A**, it enforces uniqueness (**A** + **A** = **A**).

Element ordering modifier

Generally, string elements are sorted by simple alphabetic order. For example, **aardvark** comes before **albatross**. However, you might want to store numeric data in string attributes (for example, find number). In this case, alphabetic order is not correct. For example, **20** comes before **3**. To correct this, use the **CMP_set_prop_element_order** function to set the element sort order to the **CMP_order_by_length_and_value** setting.

Element application flags

The generic compare engine has no understanding of application concepts (BOM Compare is an application built on top of the generic compare engine). However, sometimes it is necessary for the application to store modifiers on the compare descriptor. This is done using the application flags. Application flags are set by the **CMP_set_element_application_flag** function. There are three application flags:

- **BOM_compare_display_aggregate**

- **BOM_compare_stop_if_diff**

- **BOM_compare_dont_report_adds**

These are defined and documented in the **bom_tokens.h** file.

Object output order

If you have multiple primary elements, the order in which you define the elements has an impact on the order in which objects are output in a report. Consider a compare where you have two primary elements: **Item Id** and **Find Number**. Note that these are the primary elements for two of the standard compare modes. If you define the **Item Id** element before the **Find Number** element, then objects are output sorted by **Item Id** and then by **Find Number** within that.

Consider three BOM lines with the following attributes:

- **item=widgetY, seqno=10**

- **item=widgetX, seqno=20**

- **item=widgetX, seqno=30**

If you define your elements in the order **(Item Id, Seq No)**, compare outputs the lines in the order 2, 3, 1. This order is clearly not ideal. You normally would really like the output to be in the same order in which it appears in Structure Manager (in other words, 1, 2, then 3). To do this, simply reverse the order of element definition to be **(Seq No, Item Id)**. However, this affects the column output order, described in the next section.

The use of find numbers introduces another feature. Consider this pair of BOM lines:

- **item=widgetX, seqno=20**

- **item=widgetX, seqno=100**

The order output should be 1 first and 2 second. Because find numbers are strings and not integers, a string compare is performed and alphabetically 100 comes before 20. To get around this, mark the find number element (and any others that store numbers as strings) as special with the **CMP_set_prop_element_order** function:

```
CMP_create_prop_element(new_desc, ..., &seqno_element);
CMP_set_prop_element_order(seqno_element, CMP_order_by_length_and_value);
```

This call tells the compare engine to sort by length, before sorting by value.

Column output order

By default, report output is generated with the columns in the same order as the elements were defined in. In the previous section, the **Find Number** element is defined ahead of the **Item Id** element to force the object output order to match the Structure Manager display order. The side effect of this is that the report columns are output in the **(Seq No, Item Id)** order. This does not match the standard Structure Manager column display order where you expect to have the item defined in the leftmost column, followed by the find number. You can suppress the display of the find number element and define a display element against the find number property. However, the better alternative is to define a display order. To do so, create a tag array and populate it with the element tags in the order that you want the columns to appear. Then pass that array into the **CMP_set_element_display_order** function as shown in the following code:

```
tag_t seqno_element;
    tag_t itemid_element;
    tag_t elements[2];
    CMP_create_prop_element(new_desc, ..., &seqno_element);
```

```
CMP_create_prop_element(new_desc, ..., &itemid_element);
elements[0] = itemid_element;
elements[1] = seqno_element;
CMP_set_element_display_order(new_desc, 2, elements);
```

Methods

When using property-based compare elements, the system provides default behavior for comparing the property values. There are also some simple modifiers that can be applied to handle certain special cases. If none of the modifiers do what you need, you can write code that does exactly what you want and then register it against the element.

There are six method types supported by the generic compare engine:

- **Compare object**

  This method must compare a given object against a given compare set and return its opinion on whether that object belongs in that set. Note that this is just one primary compare element's opinion. Other primary elements may disagree. All primary elements must say yes for the object to be allowed into the set.

- **Compare aggregate**

  This method must compare the left and right sides of the given compare set and return its opinion on whether the two sides are equivalent. Other aggregate elements may disagree.

- **Cache**

  This method must update the element's cache when the supplied object is added to the given side of the compare set.

- **UIF name**

  This method returns the display name for this element.

- **UIF value**

  This method returns the display value for this element.

- **Free cache**

  One of the supported cache types is **CMP_cache_pointer**. This cache type means that the user is taking responsibility for creating and managing the cache. The generic compare engine simply maintains a void ***** pointer to the cache. When using this cache type, you must supply a **Cache** method (to create and update the cache), a **UIF Value** method (to return the display value for the cache), and a **Free Cache** method. This **Free Cache** method is used to release the memory allocated by the **Cache** method.

Nonproperty elements

This is the logical conclusion of the use of compare methods. A nonproperty element is an element that is entirely driven through methods. It has no standard behavior. This requires more effort, but allows maximum flexibility. For example, you could perform a compare using elements that live entirely outside Teamcenter. It could pull in data from a different database or from the Internet, which may take a significant amount of time. If you plan to do this, Siemens Digital Industries Software highly recommends caching.

For an example of a nonproperty element, see the quantity elements in the **smp_user_bom_cmp.c** file. While quantity is a BOM line property, Compare cannot handle it directly because of the need to cope with special concepts like **as required** and **undefined** (Structure Manager's default quantity, in other words, one unit).

Advanced execution concepts

One of the advantages of the generic compare engine is that it provides access to the internal data structures of the engine. This allows you to provide new output destinations and formats directly off the raw data without having to post-process one of the existing outputs. To use this, you must understand the internal data structure.

The top level object is the BOM Compare itself. This is the object that most of the BOM Compare ITK functions deal with. A BOM Compare is described as having a mode, an output destination, and two root BOM lines (often referred to as two sides of the compare, left and right).

In single and lowest-level compare traversals, the BOM Compare object has a single BOM Compare engine object. In multi-level compare traversals, the BOM Compare has a tree of nested BOM Compare engines.

A BOM Compare engine contains a list of compare sets. Each compare set contains two lists of BOM lines (one list for each side of the compare). Every BOM line in a compare set has the same values for their primary elements. Compare sets are capable of caching compared values. Primary and display elements are stored in a single cache (per element) on the compare set. Aggregate elements require a cache for each side of the compare.

To traverse this data structure, use the **BOM_compare_visit_engine** function, as shown in the following code. This function traverses the internal data structure invoking callbacks as it goes:

```
{
    tag_t compare;
    BOM_compare_create(&compare);
    BOM_compare_execute(compare,
                        bomline1, bomline2,
                        "example mode",
                        0 /* No output - we're going to do it ourselves
*/ );
    BOM_compare_visit_engine(compare,
                             enter_engine,
```

```
                                            leave_engine,
                                            visit_set,
                                            NULL  /* User data pointer */ );
}
int enter_engine(tag_t bomcompareengine, tag_t compareset, int depth,
                                                          void
*user_data)
{
    tag_t root1, root2;
    BOM_compare_ask_engine_root_bomlines(bomcompareengine, &root1,
&root2);
    printf("Entering compare engine %x, comparing BOMs %x and %x\n",
                                              bomcompareengine, root1,
root2);
    /* You could use the CMP ITK to query the contents of the engine */
    return ITK_ok;
}
int leave_engine(tag_t bomcompareengine, tag_t compareset, int depth,
                                                          void
*user_data)
{
    printf("Leaving compare engine %x\n", bomcompareengine);
    /* You could use the CMP ITK to query the contents of the engine */
    return ITK_ok;
}int visit_set(tag_t compareset, int depth, void *user_data)
{
    /* You can use the CMP ITK to query the contents of the set */
    printf("Compare set %x\n", compareset);
    /* List which compare elements changed in this set */
    int n;
    tag_t *elements;
    tag_t *bomlines;
    int i;
    CMP_ask_diff_elements(compareset, &n, &elements);
    printf("Changed elements: ");
    for (i=0; i<n; i++)
    {
        int element_type,
        int priority,
        int cache_type,
        char *uif_name,
        logical suppress_value;
        char *lvalue;
        char *rvalue;
        CMP_ask_element_info(elements[i],
                             &element_type, &priority, &cache_type,
                             &uif_name, &suppress_value);
        CMP_ask_element_value(compareset, CMP_LHS, elements[i], &lvalue);
        CMP_ask_element_value(compareset, CMP_LHS, elements[i], &rvalue);
```

```
        print("  %s has changed from %s to %s\n", uif_name, lvalue,
rvalue);
        MEM_free(uif_name);
        MEM_free(lvalue);
        MEM_free(rvalue);
    }
    MEM_free(elements);
    /* List the bomlines on LHS of this set */
    CMP_ask_objects_in_set(compareset, CMP_LHS, &n, &bomlines);
    printf("  Affected BOMLines on left hand side: ");
    for (i=0; i<n; i++)
    {
        printf("%x ", bomlines[i]);
    }
    printf("\n");
    MEM_free(bomlines);
    return ITK_ok;
}
```

Performing a compare

The BOM Compare function **BOM_compare_execute** supersedes the older **BOM_compare** call. The **BOM_compare_execute** function takes the tags of the two BOM lines to be compared, along with the compare mode, and the output type, and may optionally be given a compare context.

A compare context (gained from the **BOM_compare_create** function) is a slot into which the results of a BOM Compare are put. Each one can only hold the results of one compare. The **BOM_compare_execute** function can be told to use the default compare context by passing in a null tag.

The default compare context is the one used by the user interface when running BOM Compare from Structure Manager. If you use the default context while the user interface might be using it, you clear any BOM Compare-related BOM line highlighting and the ability to click on a report line and have the Structure Manager select the correlate BOM lines. Merely using the default context causes this, even if you do not ask for BOM line highlighting or text reports.

The standard mode names (**BOM_std_compare_*_name**) are available as manifest constants from the **bom_tokens.h** file; the output selectors (**BOM_compare_output_***) are also available from the **bom_tokens.h** file. These identifiers are used thus:

```
BOM_compare_execute( bomcomparecontext,
bomline1,
bomline2,
BOM_std_compare_single_level_name,
BOM_compare_output_bomline | BOM_compare_output_report );
```

For your own modes, you should establish and use your own manifest constants. The standard compare modes are described in *Structure Management on Rich Client*, but you can also declare new ones using the **BOM_compare_define_mode** function. To define your own modes, you need to understand compare descriptors, which are part of the generic compare engine.

The generic compare engine provides a way to compare two sets of objects, and obtain the differences between the sets. Each set is divided into subsets containing objects that match each other according to some user-defined criteria (primary keys). If both sets each have a subset whose objects match by their primary keys, then the objects within each subset are compared according to further user-defined criteria (aggregate keys), and the results for this second compare between each correlate pair of subsets are made available. A subset with no correlate subset is an addition (or a deletion). The keys are specified with a compare descriptor.

For example, the functionality of BOM Compare has been re-implemented on top of this generic engine. The two sets are the two sets of BOM lines for comparison; the BOM lines are gathered into subsets according to item ID and optionally find number, the primary keys, and where both sides of the compare has correlate subsets, they are compared for quantity and revisions, the aggregate keys.

BOM Compare output

User exits

BOM Compare can call a number of user exit functions for both ITK- and Structure Manager window-invoked compares. The user exits are:

- **USER_bom_cmp_start_report**

- **USER_bom_cmp_enter_report**

- **USER_bom_cmp_item_report**

- **USER_bom_cmp_parent_report**

- **USER_bom_cmp_exit_report**

- **USER_bom_cmp_end_report**

These user exit functions are always called by the Structure Manager window-invoked compare. User exit output is optional for the ITK compare.

A number of ITK functions are available to allow the user exit functions to query the compare results:

- **BOM_compare_list_bomlines**

- **BOM_compare_ask_differences**

- **BOM_compare_ask_qty**

- **BOM_compare_ask_rev**

- **BOM_compare_ask_seqno**

This list of functions has been augmented by the generic compare functions that can be applied to compare engines.

BOM line output

The following BOM line string property is set to contain the changes relating to the particular BOM line:

```
bl_compare_change
```

The following two properties are additionally supported when the **BOM_compare_legacy_properties** preference is true:

```
bl_quantity_change
bl_revision_change
```

If no differences are found for a particular BOM line, these properties are all blank.

If the **BOM_compare_legacy_properties** preference is false or unset:

- **bl_compare_change**

  Contains the type of change for this line, either Added or one or more of the Aggregate keys and key values for the BOM Compare Mode specified. For the standard compare modes, the keys are Qty or Rev for quantity changes and revision changes, respectively, so this property might look like, Qty:1→2 or Rev:A→B. This is a comma-separated list if more than one change occurs (for example, both quantity and revision changes might look like Qty:1→2,Rev:A→B).

If the **BOM_compare_legacy_properties** preference is true:

- **bl_compare_change**

  Contains the type of change, Added, Qty, or Rev for additions, quantity changes and revision changes, respectively. This is a comma-separated list if more than one change occurs (for example, both quantity and revision changes might look like, Qty,Rev).

- **bl_quantity_change**

  Contains details of a quantity change (in other words, if Qty is a substring of the **bl_compare_change** attribute). The change is stored as the two quantities separated by an arrow (for example, 1→2).

- **bl_revision_change**

Contains details of the revision change (in other words, if Rev is a substring of the **bl_compare_change** attribute). The change is stored as the two revisions separated by an arrow (for example, A→B). If more than one revision of an item exists in one of the BOMs then all the revisions are listed (for example, A,B→C).

These properties may be cleared by calling the **BOM_compare_clear** function.

> Note:
>
> If another compare is started (in a particular compare context) before the previous one has been cleared, the old properties are automatically cleared.

Report output

Report output can be retrieved with the **BOM_compare_report** function. Note that you must request report output when you call the **BOM_compare_report** function, otherwise no report is generated.

The **BOM_compare_report** function returns the report as an array of formatted character strings. The first string contains the column headers for the report. The function also returns an array of compare items that match the report lines. These compare items can be queried with the following functions:

- **BOM_compare_list_bomlines**

- **BOM_compare_ask_differences**

- **BOM_compare_ask_seqno**

These functions have been augmented by the generic compare functions, as for the user exits.

These functions are available only if the **BOM_compare_legacy_report** preference is true:

- **BOM_compare_ask_qty**

- **BOM_compare_ask_rev**

- **BOM_compare_ask_seqno**

Report lines that do not have matching compare items (for example, the column header line) have their compare item listed as **NULLTAG**.

BOM Compare output suppression

BOM Compare output suppression allows the BOM Compare user exit functions to switch off certain forms of compare output. For example, the default user exit functions do not perform any useful actions. To prevent the Structure Manager window-based compare from calling unnecessary user exit

functions, the very first user exit function (**USER_bom_cmp_start_report**) suppresses all other user exit output, using the following line of code:

```
BOM_compare_suppress (line1, BOM_compare_output_userexit);
```

This suppression lasts for the duration of the current compare. The primary intended use for this facility is if you wish to replace the BOM Compare's internal report window with your own report. To achieve this, you must write the user exit functions to generate the report and display it in a window.

Once the report generation code has been written, the problem is how to control which report window is displayed and when. You must first suppress the user exits when report output was not selected in the **BOM Compare** dialog box:

```
If:
( (output & BOM_compare_output_report) == 0)
  {
    BOM_compare_suppress (line1, BOM_compare_output_userexit );
    return ITK_ok;
  }
```

You should now suppress the internal report window:

```
BOM_compare_suppress ( line1, BOM_compare_output_report );
```

There are many other potential uses for output suppression. Some further examples are given in the **USER_bom_cmp_start_report** user exit source code.

BOM Compare visitor

After you run a compare (possibly without any output modes specified), you can ask the system to walk the results tree, running user-supplied callbacks at each node. This is very similar to the user exits system above, but because you pass in the three callback functions instead of linking them into the user exits library, you can process the results in various ways without needing to change the user exits library. The callbacks have this correlation with the user exits:

- The **visit_set** function is identical to the **USER_bom_cmp_item_report** function, except that it has the extra **user_data** parameter - a void ***** parameter supplied by your call to the **BOM_compare_visit_engine** function, pointing to a structure you want your callbacks to build or maintain.

- The **enter_engine** function is analogous to **USER_bom_cmp_parent_report** function. The following code makes that relationship explicit:

```
#include <unidefs.h>
#include <tc/tc.h>
#include <bom/bom.h>
#include <user_exits/user_exits.h>
int enter_engine( tag_t compare_engine, tag_t compare_set, int depth, void * user_data)
```

```
{
    int itk_ret;
    tag_t bomline1;
    tag_t bomline2;
    if ( ( itk_ret = BOM_compare_ask_engine_root_bomlines( compare_engine, &bomline1,
      &bomline2) ) != ITK_ok )
        return itk_ret;
    return USER_bom_cmp_parent_report( bomline1, depth );
}
```

- The **leave_engine** routine is called when all descendants of the latest active parent announced through the **enter_engine** function have been visited. All ordering is depth-first recursive, so each parent node is only reported once immediately before all of its descendants, and once immediately after all of its descendants. The following code counts the number of BOM lines (on the left-hand or right-hand side) determined to be different by the compare:

```
/* Count BOM Lines in a compare, declares
extern int count_bomlines_in_compare ( tag_t compare_context, int cmp_side );
*/
#include <bom/bom.h>
#include <fclasses/cmp.h>
#include <tc/tc.h>
/* Structure for passing nominated side and current count around */
typedef struct
{
    int count;
    int side;
} count_bomlines_t;
/* Callback to add the number of objects for each set to user_data-count. */
/* Note, it would be trivial to put a loop in here to examine/modify
the BOM Lines in bomlines[], instead of merely counting them */
static int count_bomlines_visit_set ( tag_t compareset, int depth, void * user_data )
{
    count_bomlines_t *data = (count_bomlines_t*) user_data;
    int count;
    tag_t *bomlines;
    {
        /* Ask compareset for the bomlines on the nominated side. (I'm
        only interested in how many there are, not what they are.) */
        int itk_ret;
        if ( ( itk_ret = CMP_ask_objects_in_set( compareset, data->side, &count,
          &bomlines ) ) != ITK_ok )
            return itk_ret;
    }
    data->count += count;
    MEM_free( bomlines );
    return ITK_ok;
}
/* Run over the BOM compare results passed in, and return the number of
BOM lines, or -1 on error. */
extern int count_bomlines_in_compare ( tag_t compare_context, int cmp_side )
{
    count_bomlines_t results = { 0, cmp_side };
    /* Use NULL for parent (engine) entry/exit callbacks, because
    they're not interesting here */
    if ( BOM_compare_visit_engine( compare_context, NULL, NULL, count_bomlines_visit_set,
      (void*) &results ) != ITK_ok )
    {
```

```
        return -1;
    }
    return results.count;
}
```

Product Structure (PS) module

Introduction to the Product Structure (PS) module

The Product Structure (PS) ITK module manages the creation, manipulation and storage of product structure data within Teamcenter.

The PS module handles product structure operations at a lower level. It is also responsible for the creation and manipulation of the basic structure objects (in other words, BOM views and occurrences).

This module is most useful for large scale import of structure data into Teamcenter or batch operation to update specific structure attributes where the overhead of building up the full presentation of BOM-related information is not required.

Product structure in Teamcenter is represented by assembly items containing links to component items which make up the assembly. The structure of items is represented in BOM view objects. The links to component items are known as occurrences.
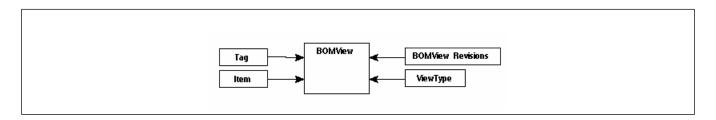
The PS ITK module covers the creation of BOM views and BOM view revisions for items and item revisions, the creation and modification of occurrences to reference the component parts of an assembly, and the attachment of structure-related attributes to occurrences.

Product Structure classes

BOM view

When an item is an assembly of other items, its assembly structure is represented by a BOM view. A BOM view is a workspace object. It is an object distinct from the item in order to support multiple views functionality, when a separate BOM view is required to represent each different view of an item's structure.

The class attributes and methods for BOM view in the PS module are shown in the following figure:

BOM view class attributes and methods

In the figure:

- Item is the item to which this BOM view belongs, inquired by the **PS_ask_item_of_bom_view** function.

- BOM view revisions are all of the revisions of this BOM view. They can be listed using the **PS_list_bvrs_of_bom_view** function.

- **Viewtype** specifies the type of this BOM view with the tag of a **View Type** object. Use the **PS_ask/ set_bom_view_type** functions to inquire or modify its value, respectively.

- BOM view also inherits name and description from the workspace object, which can be inquired and changed using the appropriate **WSOM** functions.

  A BOM view is created by the **PS_create_bom_view** function. The name of the BOM view must be unique for all BOM views of the parent item. Be aware that no initial revision is created, this is done by a call to the **PS_create_bvr** function.

  > Note:
  >
  > A new or modified BOM view is not saved to the database until you make a call to the **AOM_save** function. The item it is placed in is also modified and must also be saved with a separate call to **AOM_save**.

BOM view revision

The structure of an assembly item may change between successive revisions of the item. Therefore the actual structure information of an item revision is stored in a BOM view revision referenced by that item revision. A revision of BOM view "A" must specify the structure of a revision of the item that owns BOM view "A".
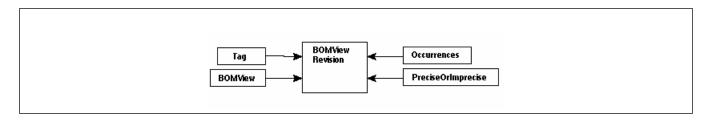
There is no one-to-one correspondence between item revisions and BOM view revisions, because where two or more revisions of an item have the same structure they may share references to the same BOM view revision. A new working item revision based on a previous released revision retains a reference to the released revision's BOM view revision until such time as the structure of the working item revision needs to be modified. An item revision may reference only one revision of a BOM view at a time.

BOM view revisions are workspace objects. They are presented in the workspace separately from item revisions for three reasons:

- To allow access controls to be placed on the structure of an item revision independently of access controls on the rest of the item revision's data. For example, to protect the structure from further modification while detailed changes are still being made to a drawing.

- To allow the structure of an item revision to be put through a release procedure independently of the rest of the data for an item revision.

- To allow for multiple views functionality, when an item revision is able to reference BOM view revisions of different BOM views, each of which represents a distinct view of the structure of the item.

By default a BOM view revision is shown in the workspace as a specification of an item revision.

The class attributes and methods for BOM view revision in the PS module are as shown in the following figure.

BOM view revision class attributes and methods

The occurrences of a BOM view Revision are listed by the **PS_list_occurrences_of_bvr** function. A BOM view Revision's occurrences are either all precise, in which case they reference child item revisions or they are all Imprecise, in which case they reference child items. This is determined by the state of the **PreciseOrImprecise** flag. This flag may be inquired with the **PS_ask_is_bvr_precise** function and changed with the **PS_set_bvr_precise/imprecise** functions.

> Note:
>
> The **PS_set_bvr_precise** function does not set view pseudo folder contents to **ItemRevisions**. You need to use the **BOM_line_set_precise** function instead. If you have the following code in your ITK program:
>
> ```
> PS_set_bvr_precise(bvr);
> AOM_save(bvr);
> ```
>
> Replace it with the following:
>
> ```
> BOM_create_window(&window);
> BOM_set_window_config_rule( window, rule);
> BOM_set_window_pack_all (window, TRUE);
> BOM_set_window_top_line(window, NULLTAG, item_rev, bvrs, &top_line);
> BOM_line_set_precise(top_line, TRUE);
> BOM_save_window(window);
> BOM_close_window(window);
> ```

The BOM view of which it is a revision is obtained by calling the **PS_ask_bom_view_of_bvr** function.

A new BOM view revision is created using the **PS_create_bvr** function, giving the tag of the parent BOM view and the item revision in which it is to be placed.

The **PS_revise_bvr** function creates a new BOM view revision based on an existing revision of the same BOM view. The **PS_copy_bvr** function creates a new BOM view revision based on an existing revision of a different BOM view.

> **Note:**
>
> A new or modified BOM view revision is not saved to the database until you make a call to the **AOM_save** function. This also applies when occurrences of a BOM view revision are modified. Additionally, when a new BOM view revision is created, the item revision it is placed in is also modified and must also be saved with a separate call to the **AOM_save** function.

Occurrences

The single-level structure of an item revision is comprised of a collection of links from a BOM view revision (the parent) to the items that are components of the assembly (the children). These links are known as *occurrences*. An occurrence represents the usage of an item or an item revision within the product structure of a parent. You can distinguish between different kinds of occurrences in the product structure by referring to occurrence types and occurrence sequencing in the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is available on Support Center.)

> **Note:**
>
> Occurrence is not presented as a distinct class within Teamcenter, rather as an attribute of BOM view revision. In the PS ITK module, occurrences are always addressed through their parent BOM view revisions.

Creation, modification or deletion of an occurrence is treated as a modification of its parent BOM view revision. This allows access control to be centralized on the BOM view revision. You do not have the danger of two users making simultaneous changes to different occurrences within the same BOM view revision.

Following are characteristics of occurrences:

- Attaching attribute data

  Attribute data can be attached to occurrences. This is data that is specific to a particular usage of a component item within an assembly, for example usage notes and find numbers. Data that is generic to all usages of an item should be attributes of the item itself.

- Precise/imprecise

  If the child (component part) of an occurrence is a specific item revision, the occurrence is precise. If the occurrence is a generic reference to an item, it is imprecise. If you have an imprecise occurrence, the item it references may be resolved at run time to an exact item revision by the application of a configuration rule.

- Multiple view functionality

  To support multiple views functionality, an occurrence also specifies a child BOM view to allow a particular view of a sub-assembly item to be included in the structure.

- Multi-level product structure

  A multi-level product structure is built up by BOM view revisions having occurrences referencing other items with BOM views and so on.
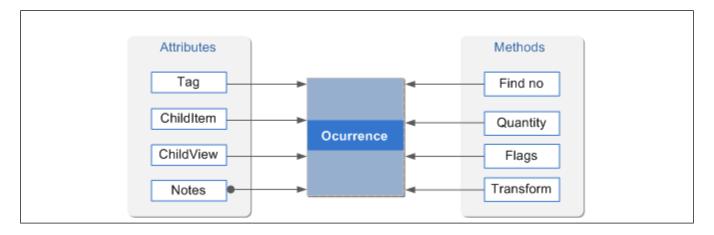
- Substitutes

  Rather than referencing a single component item, an occurrence can contain links to several substitute items. Substitutes are interchangeable with one other in the assembly. However, one of these substitute is specified as the preferred substitute. All substitute of a single occurrence share the same attributes, the attributes specified for the preferred substitute.

  Any modifications the occurrence substitutes is a modification to the BOM view revision. Permanent modifications require write access to the BOM view revision. Temporary modifications may be made during a Teamcenter session but cannot be saved to the database.

Class attributes and methods

The class attributes and methods for occurrence in the PS module are shown in the following figure.

Occurrence class attributes and methods

Occurrences are not presented as separate objects in the ITK interface. You must always address an occurrence in the context of its parent BOM view revision using the tag of the BOM view revision as well as the tag of the occurrence itself. When you modify an occurrence of a BOM view revision, you modify that BOM view revision. The change is not written to the database until you save the parent BOM view revision using the **AOM_save** function.

Occurrences are created with the **PS_create_occurrences** function. The **ChildItem** is set to the tag of a component item revision or item, depending on whether the parent BOM view revision is precise or imprecise. The **ChildView** can be set to the tag of a BOM view of the **ChildItem** or to **NULLTAG**.

You can inquire about the child of an occurrence with the **PS_ask_occurrence_child** function or change it with the **PS_set_occurrence_child** function. When **PS_set_occurrence_child** is called, only the child

component of the occurrence is changed. All of the occurrence attributes (find number, notes, and so on) are retained. This is the functionality employed by the **Replace** command in the Structure Manager user interface.

An occurrence may have brief textual notes attached to it. A note is a string of up to 160 characters. Each note has a type defining its purpose (for example, usage, color, adjustment). An occurrence may have any number of notes, provided each is of a different type. The type must be one of a set of **Note Type** objects defined by the system administrator for a Teamcenter installation. You can access and edit notes using the **PS_list_occurrence_notes**, **PS_ask/set_occurrence_note_text**, and **PS_delete_occurrence_note** functions.

A find number is used to identify and sort individual occurrences within a single level of structure. It is a 15-character alphanumeric string and is inquired or changed using **PS_ask/set_seq_no**, respectively.

Occurrence has a quantity attribute containing the amount of the child items specified by the occurrence (for example, 4 bolts or 1.5 liters of oil). Quantity is stored as a real number and is accessed by the **PS_ask/set_occurrence_qty** functions. A negative quantity means the quantity is undefined.

> Note:
>
> An occurrence quantity must be expressed in the unit of measure attached to the child (component) item referenced by the occurrence. Where a component item has no unit of measure, the quantity value is interpreted as the number of component items referenced by the occurrence. An occurrence which specifies more than one usage of a component item is known as an aggregate occurrence.
>
> Additional information about defining units of measure can be found in the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is not available in PDF format. It is available only in the Teamcenter HTML help).

An occurrence also carries a set of flags for storing additional data. Only one flag attribute is implemented, **PS_qty_as_required**. When set, it indicates the child item is to be used as required. Flags are inquired using the **PS_ask_occurrence_flag** function and changed using the **PS_set/clear_occurrence_flag** functions.

A 4x4 transformation matrix can be stored on an occurrence for passing positional information to a CAD system and is accessed using the **PS_ask/set_plmxml_transform** functions.

By default, the occurrences of a BOM view revision are listed in the order in which they were created. This order can be altered using the **PS_move_occurrence_to/up/down** functions.

> Note:
>
> A change of ordering is treated as a modification to the parent BOM view revision. Call the **AOM_save** function to store the change in the database. Additionally, this ordering functionality is not available through the Structure Manager user interface. Occurrences are sorted by find number unless a different custom sort algorithm is supplied for a Teamcenter installation.

You can create a substitute with the **PS_add_substitute** function. It takes an existing item and makes it a substitute to the existing child of an occurrence. If the occurrence previously had no substitutes, the existing child is used as the preferred substitute.

You can delete a nonpreferred substitute with the **PS_delete_substitute** function. The preferred substitute cannot be deleted.

You can make a nonpreferred substitute the preferred substitute using the **PS_prefer_substitute** function. If the BOM view revision is write-protected or frozen, the change to the preferred substitute succeeds, but the change cannot be saved to the database. This case is flagged by the **is_temporary** return value.
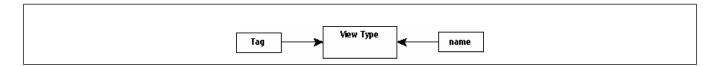
You can find the occurrence substitutes using the **PS_ask_has_substitutes** and **PS_list_substitutes** functions. The **PS_ask_has_substitutes** function returns true if the occurrence has two or more substitutes defined. The **PS_list_substitutes** function returns a list of the occurrence substitute.

View type

Each BOM view in Teamcenter has a type attribute. A **View** type is an object that defines an identifying string for a type of view (for example, design view or manufacturing). The system administrator defines **View** types for a Teamcenter installation.

Teamcenter provides a default **View** type. All BOM views created via the Teamcenter user interface are of this type. The name of this **View** type is defined by the **PS_default_view_type_name** token in the **ps_tokens.h** header file .

The class attributes and methods for the **View** type in the PS module are shown in the following figure.

View type class attributes and methods

A **View** type is a site-defined classification of BOM views. You can get a list of all **View** types defined for an Teamcenter installation from the **PS_view_type_extent** function. You can search for a particular **View** type by name using the **PS_find_view_type** function. You can find the name of a **View** type object from its tag by calling the **PS_ask_view_type_name** function.
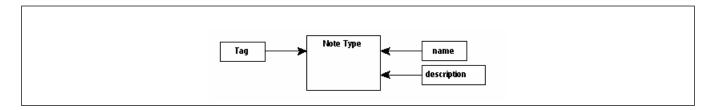
> Note:
>
> Only the system administrator can alter the list of available **View** types, using the **PS_create_view_type** and **PS_delete_view_type** functions.

Note type

You can attach text notes to occurrences. Each note has a type (for example, usage or adjustment), which defines the purpose of the note. The possible types of notes are determined by the set of **Note** type objects defined for a Teamcenter installation by the system administrator.

The class attributes and methods for Note Type in the PS module are shown in the following figure:

Note type class attributes and methods

Note type is a list of types of occurrence note, as defined for a Teamcenter installation by the system administrator. Each **Note** type has a short name used as an identifier (for example, at the head of an attribute column in Structure Manager) and a description which can be used for a longer explanation of the purpose of the **Note** type.

Call the **PS_note_type_extent** function to get a list of available **Note** types. To find individual types by name, use the **PS_find_note_type** function.
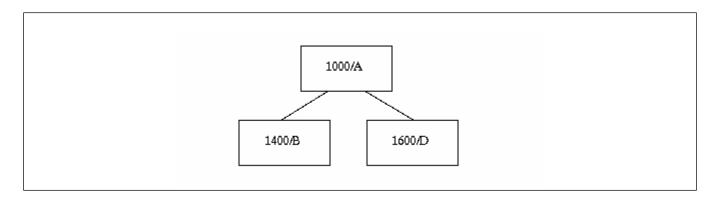
Loading of PS objects

When a call to a PS ITK function needs to access the attributes of an object, that object (if not already loaded) is loaded automatically. When a call to a PS ITK function modifies the attributes of an object, that object is locked for modification automatically if it is not already locked.

Adding attributes

You can add attributes to PS objects using the PS ITK function. Each class within the PS module can have a number of client data attributes defined for it. Each instance of such a class may then have each of these client data attributes set to reference a POM object. This POM object must be of the class specified in the definition of the appropriate client data attribute.

Product Structure model

When Teamcenter displays the product structure of an assembly, you see a tree of item revisions. The following figure shows an example:

Product structure model

This simplistic model is complicated by two additional requirements:

1. Teamcenter does not store references to exact revisions of items. This means the assembly would have to be modified every time you wanted to use a new revision of a component. Instead Teamcenter stores a reference to an item and turns this into an exact item revision at run time by application of a configuration rule.

    The PS module itself does not deal with the use of configuration rules, but configuration impacts the PS module with the requirement to store references to imprecise items as well as precise item revisions.

2. PS has been designed to facilitate multiple views functionality. This is done by using the BOM view object, each of which represents one view of the structure of the item.

In an imprecise case, a structure references an item as well as a BOM view of that item. At run time, a configuration rule is applied to find the appropriate revision of the item. If this component item revision references a revision of the specified BOM view, then this component is a subassembly whose structure is defined by the BOM view revision found. Otherwise, this component is treated as a piece part (it is not further divisible into a substructure). In the precise case, the item revision is referenced directly by the occurrence, therefore the configuration step is eliminated.

Where used

The PS ITK module supplies three functions to produce where used reports on item revisions. Each reports parent item revisions having a BOM view revision with occurrences referencing the given item revision up to a specified number of levels.

*   **PS_where_used_all**

    This function reports all usages regardless of precise/imprecise status and configuration rules.

*   **PS_where_used_configured**

    This function applies a configuration rule (from the CFM module) to report usage in context.

- **PS_where_used_precise only**

  This function looks at precise occurrences.

Relationship to the BOM ITK module

The BOM ITK module sits at a level above the PS module and beneath the Structure Manager user interface. It is oriented toward the presentation of and interaction with product structure and encompasses the application of configuration rules.

The BOM module pulls together information from items, item revisions, BOM view revisions and occurrences to present them as entries in a bill of materials. The majority of product structure work can be done using the BOM module with the exception of creating new BOM views. For this, PS must be called.

The PS ITK module is most useful for lower level operations (such as the large scale import of structure data into Teamcenter) or batch operations to update specific structure attributes where the overhead of building up the full presentation of BOM-related information is not required. Direct calls into the CFM ITK module can provide configuration information to support use of the PS in this way without needing to use the BOM module.

Product Structure Traversal Engine

Basic features of the Traversal Engine

The Traversal Engine (TE) is a module that is distributed as a shared library and as callable subroutines (TE API calls). Typical common tasks include:

- Generate reports

- Release assemblies

- Transfer ownership of Teamcenter objects that constitute the product structure

The Traversal Engine manages the traversal, while you are only required to develop simple ITK programs. The Traversal Engine allows you to register these functions at different stages of the traversal. These functions are executed at each node and contain the information of the nodes, which can be processed. The core module is programmed in object-oriented programing (OOP), so the framework can be expanded by adding new classes at a later date, and can also be extended to traverse any Teamcenter structure such as product structure or folders. Presently the core module relates to product structure only. The **ps_traverse** utility, located in the *TC_ROOT*/**bin** directory has also been developed to demonstrate the use of TE.

The features of the Traversal Engine are:

- Represents the Teamcenter structure as a tree data structure with each node representing the Teamcenter object.

- Provides user exits at different stages and allows the registration of user-defined functions known as handlers and selectors.

- Allows the handlers and selectors to return statuses that determine the creation of the tree and the traversal direction.

- Allows product structure configuration through revision rules and saved variant rules during the creation of the tree.

- Allows inputs from the command line and configuration file.

- Allows you to store and retrieve a float value with a key as a string, typically used for summing up functionality.

With these features, you can develop programs that:

- Generate reports.

- Apply task release.

- Release assembly, set assembly precise, and other similar tasks

User exits

Teamcenter provides user exits to register selectors and handlers that are executed at different stages of the traversal activity, such as creation of the tree, traversing in the forward direction, traversing in the reverse direction, and so on.

Selectors and handlers are user-defined functions that need to be developed following a template. These functions return the decisions that determine the traversal direction. These are registered at different stages of the traversal and are executed during the creation and the traversal of the tree. The selectors are run during the creation of the tree nodes. These determine the values stored in each node and decide whether a particular branch needs to be traversed or not. The handlers are run during the tree traversal. The handlers have the information of the node and the information pertaining to the Teamcenter object that the node represents.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Minimum requirements

You need the following minimum requirements to run the Traversal Engine:

- The Teamcenter ITK development environment.

- To develop new programs in TE, knowledge of writing ITK programs is required.

Installation

To confirm that you have installed TE successfully, check if the following have been added to their respective directories:

- The **libitk.**_libraryextension_ library in the _TC_ROOT_/**lib** directory.

- **ps_traverse** executable in the _$TC_ROOT_/**bin** directory.

- The **ps_traverse.cfg** and **te_template_itk_main.c** files in the _TC_ROOT_/**sample/examples** directory.

Developing a traversal program

Once you custom develop your selectors and handlers by using the ITK functions and the TE API calls, you can compile them into an executable by using the **compile** and **linkitk** scripts in the _TC_ROOT_/**sample** directory.

API calls are classified into:

- Initializing and terminating calls. TE modules have to be initialized to use any TE call.

- Configure PS. These configure the PS using revision rule or saved variant. View type to traverse also can be specified.

- Register handlers and selectors. These register the provided the functions as selector or handler.

- Fill/Traverse. These create the tree and traverse the tree.

- User interface calls. These write messages to the log file and console and receive input from the configuration file and the command line arguments.

Refer to the **te.h** header file in the _TC_ROOT_/**include/itk** directory.

Sample implementation

The **te_template_itk_main.c** file in the _TC_ROOT_/**sample/examples** directory demonstrates TE functionality. It can be compiled and studied to understand how new utilities can be developed.

4. ITK customization

Sharing of data objects

Object import and export

Orientation to object import and export

You use the import/export object functionality to move data that is associated with Teamcenter objects between Teamcenter sites with either read-only or read/write privileges.

> Note:
>
> Use PLM XML import and export when possible.
>
> This functionality is in addition to the **IMF_export_file** and **IMF_import_file** routines which are used to export and import files between the Teamcenter and the operating system.
>
> The **IMF_import_file** and **IMF_fmsfile_import** ITK APIs do not validate whether the file is a text file or a binary file validation. When using these APIs, FMS does not attempt to validate the type of file against the type specified on the function call. The caller must ensure that the file for transfer is of the specified type.

Object import and export interface

There are three ways to import or export an object:

- Using the ITK functions

- Using the user interface

  Select your objects from the workspace window and pick the appropriate option (**Import** or **Export**) from the **Tools** menu. Objects are selected as follows:

  - If you select a (collapsed) folder, all objects inside that folder are exported or imported that can be exported or imported.

  - If you select many objects at one time, they can be exported or imported in a single operation.

- There are command line interfaces to execute the export or import process in batch mode for items only. They are **item_export** and **item_import**.

> Note:
>
> Imported objects are in the same folder structure as they were when exported.

4-160

Server Customization, Teamcenter 14.2
© 2022 Siemens

Object types

The object types, including all their internal supporting data, that can be imported/exported between Teamcenter sites are:

- **Folders**

  You can choose to export any general folder. You cannot export pseudo folders. Pseudo folders are folders displayed in your workspace that display objects with a specialized relationship to an item or item revision.

- **Datasets**

  When exporting, you can choose to export either all versions, the latest version, or a specific version.

- **Forms**

  When exporting, you can choose to export a form. The definition of the form must be identical at both sites.

- **Item**

  When exporting, if you choose an item, the item and all its related data (such as the item revisions, BOM view and BOM view revisions, associated item master and item revision master forms, and any exportable requirement, specification, manifestation or reference objects) are exported. Additionally, if the selected item has an assembly (structure), then all the items that make up the assembly are exported.

  You cannot choose part of an item to be exported. For example, you cannot choose an item revision alone to be exported. You need to select the item which contains this item revision, in order to export the item. Similarly, the same would be true with BOM view and BOM view revision objects.

All Teamcenter files that the dataset represents are exported, including the operating system files that these encapsulate and the dataset's revision anchor.

When you export, there is an option to transfer the ownership to another site. If the ownership is transferred to a particular site, then when the same objects are imported at that particular site, they are imported with a read/write privilege. If the ownership is not transferred, then the ownership of the objects still belongs to the export site. When these objects are imported at another site, they are imported with a read-only privilege. Any modifications attempted on them are not be allowed.

Object ownership

If the same owning-user name exists at my importing site as owned it at the exporting site, then set that user to own it here. Only if that lookup fails does it resort to the simple rule that the user who runs the import gets to own the object.

When an object is imported, if the user who exported the object exists at the importing site, the object's owner is switched from the user at the exporting site to the one at the importing site. If the user does not exist at the importing site, the owner is the user who performed the import. The group is the group in which the owner logged into to perform the import.

In general with import, object site ownership takes precedence in determining who has the right to update the object.

Export of released objects

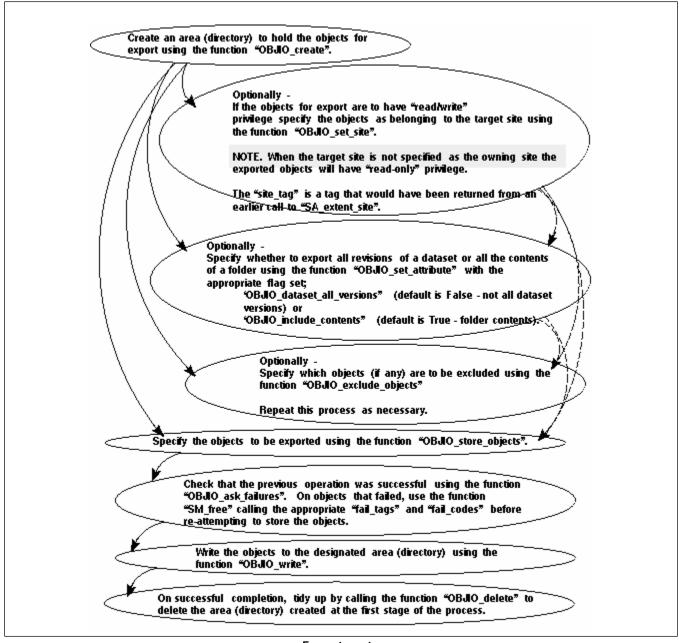The following rules apply for released objects:

- Objects that are within a release process can be exported either with read/write privileges or with read-only privileges. Access to released objects are governed by the local rule tree.

- If such an object is imported at another site, it cannot be used in a release process at that site if it was exported with read-only privileges. If it was exported with read/write privileges, then it can be used in a release process.

- If an object is locked (in other words, it has been exported with read-only privileges and is therefore locked at the receiving site) or it has been exported with read/write privileges but not re-imported (therefore locked at the sending site), it cannot be used in a release process.

> Note:
>
> All released objects are exported with their status (for example, Released to Manufacture).

Exporting objects – the route

The route of exported objects is shown in the following figure.

Create an area (directory) to hold the objects for export using the function "OBJIO_create".

Optionally -
If the objects for export are to have "read/write" privilege specify the objects as belonging to the target site using the function "OBJIO_set_site".

NOTE. When the target site is not specified as the owning site the exported objects will have "read-only" privilege.

The "site_tag" is a tag that would have been returned from an earlier call to "SA_extent_site".

Optionally -
Specify whether to export all revisions of a dataset or all the contents of a folder using the function "OBJIO_set_attribute" with the appropriate flag set;
    "OBJIO_dataset_all_versions" (default is False - not all dataset versions) or
    "OBJIO_include_contents" (default is True - folder contents).

Optionally -
Specify which objects (if any) are to be excluded using the function "OBJIO_exclude_objects"

Repeat this process as necessary.

Specify the objects to be exported using the function "OBJIO_store_objects".

Check that the previous operation was successful using the function "OBJIO_ask_failures". On objects that failed, use the function "SM_free" calling the appropriate "fail_tags" and "fail_codes" before re-attempting to store the objects.

Write the objects to the designated area (directory) using the function "OBJIO_write".

On successful completion, tidy up by calling the function "OBJIO_delete" to delete the area (directory) created at the first stage of the process.

Export route

Responsibilities and restrictions when exporting

The following enable you to determine the responsibilities and restrictions when exporting:

- It is the responsibility of the user who is exporting objects to inform the system manager which directories need to be copied and to which site.

- It is the responsibility of the system manager to set up the list of other sites which are known to the local site.

- It is the responsibility of the system manager to send directories of the exported objects to the receiving sites, (for example, using tape, ftp, and so on).

The following significant object types cannot be exported:

- Users.

- Other system administration objects.

Import semantics

The importing of an object is straightforward unless the same object already exists at the importing site. In this case, the behavior of the objects is described as follows:

- Revisioned objects – datasets

  Although datasets do not need to have unique names, they do have unique internal identifiers. Therefore, when a dataset is imported, the system knows whether it already exists. If the dataset does exist which is the same revision, no import takes place.
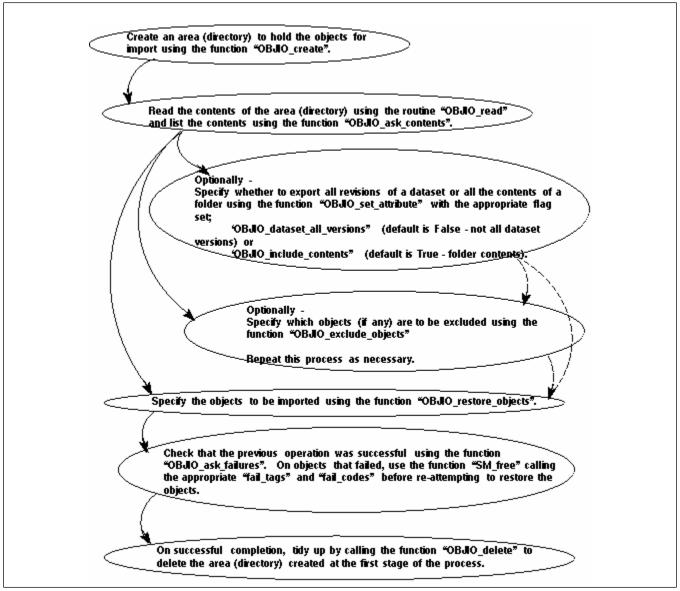
- Revisioned objects – item

  Items need to have unique ID's. When an item is imported, be aware of the following issues:

  - It is the responsibility of the user who is exporting objects to inform the system manager which directories need to be copied and to which site.

  - If an item with the same ID does not exist in the database, then an item with that ID is created and owned by the user doing the import.

  - If an item with the same ID exists in the database, then the requirement, manifestation, reference and specification objects of the importing item are added on to the existing item.

- Nonrevisioned objects – folders

  These do not have unique names, therefore there are no problems when importing objects.

Importing objects – the route

The route of imported objects is shown in the following figure.

Import route

Create an area (directory) to hold the objects for import using the function "OBJIO_create".

Read the contents of the area (directory) using the routine "OBJIO_read" and list the contents using the function "OBJIO_ask_contents".

Optionally -
Specify whether to export all revisions of a dataset or all the contents of a folder using the function "OBJIO_set_attribute" with the appropriate flag set;
"OBJIO_dataset_all_versions" (default is False - not all dataset versions) or
"OBJIO_include_contents" (default is True - folder contents).

Optionally -
Specify which objects (if any) are to be excluded using the function "OBJIO_exclude_objects"
Repeat this process as necessary.

Specify the objects to be imported using the function "OBJIO_restore_objects".

Check that the previous operation was successful using the function "OBJIO_ask_failures". On objects that failed, use the function "SM_free" calling the appropriate "fail_tags" and "fail_codes" before re-attempting to restore the objects.

On successful completion, tidy up by calling the function "OBJIO_delete" to delete the area (directory) created at the first stage of the process.

Restrictions when importing

The following restrictions apply when importing an object:

- An imported object cannot be modified (but may be copied), unless it is exported with read/write privileges.

- When an item is imported read-only, then the following rules apply:

  - Existing requirements and specifications cannot be deleted, and new requirements and specifications cannot be added.

- Existing manifestations and references can be cut, and new manifestations and references can be added.

- Attributes on the imported objects cannot be modified.

- The item can be deleted from the database only if the item itself is selected. You cannot delete any of the item revisions alone.

- The BOM view and BOM view revisions cannot be modified.

- When an item is imported with read/write privileges (with site ownership), then the following rules apply:

  - The item cannot be deleted from the database and none of its requirements, specifications, manifestations and references can be removed. This is because another site may have a reference to it.

  - New requirements, specifications, manifestations and references can be added.

  - Attributes can be modified.

  - If the item has been previously imported, the subsequent import does not delete any references, but may add new ones. This occurs so that the import does not wipe out any local references you may have added. It is important to note that references deleted at the originating site are not removed from the importing site. You may want to delete the item before re-importing it to ensure that you get an exact copy of the original.

- Objects, when imported, are given new ACLs. For instance, the owner (as well as their group) is updated to that of the importing user. However, if the object was exported as read-only, the importing user is not able to modify the object.

PLM XML ITK

PLM XML extension points

You can add user exits to customize filter rules, actions rules, import methods, export methods, and schema mapping. Develop your code and then use the Business Modeler IDE to define the extension and assign it to an extension point.

The following table shows the six PLM XML extension points available for customization in the Business Modeler IDE. Each of these user exits or extension points has one extension, in other words, the base extension.

Extension rule

	User exit name	Purpose
USER Filter Registrations	USER_register_plmxml_filters	Allows you to register custom filter rules used during import/export.
USER Register actions	USER_register_plmxml_actions	Allows you to register custom action rules used during import/export.
USER Register Export Methods	USER_register_plmxml_export_methods	Allows you to register custom export methods. The export method is registered for exporting an object out of the Teamcenter database.
USER Register Import Methods	USER_register_plmxml_import_methods	Allows you to register custom import methods. The export method is registered for importing an object into the Teamcenter database.
USER Schema Mapping Regs	USER_register_plmxml_schema_mapping	Allows you to map Teamcenter and PLM XML objects.

Using ITK and PLM XML to export a BOM view revision

The following example shows how to use ITK to export a BOM view revision with PLM XML:

```
#include <stdlib.h>
#include <tc/tc.h>
#include <tc/emh.h>
#include <tccore/item.h>
#include <itk/mem.h>
#include <pie/pie.h>
#include <bom/bom.h>
#define SampleError  1
#define EXIT_FAILURE 1
#define ERROR_CHECK(x) {              \
    int stat;                         \
    char *err_string;                 \
    if( (stat = (x)) != ITK_ok)   \
    {                                 \
       EMH_get_error_string (NULLTAG, stat, &err_string);                \
       printf ("ERROR: %d ERROR MSG: %s.\n", stat, err_string);          \
       printf ("FUNCTION: %s\nFILE: %s LINE: %d\n",#x, __FILE__, __LINE__); \
       if(err_string) MEM_free(err_string);                              \
       exit (EXIT_FAILURE);                                              \
    }                                                                    \
}
int ITK_user_main(int argc, char* argv[]);
static void initialize ( char *usr , char *upw , char *ugp);
static void display_help_message(void);
static int find_topline_of_itemid(char *itemid,char *rev, int *n_tags, tag_t **tags);
```

```c
static int export_bvr();
/* Main   */
int ITK_user_main(int argc, char* argv[])
{
    int  status = 0;
    if ( argc <= 6 )
    {
        display_help_message();
        return( SampleError);
    }
    char *usr = ITK_ask_cli_argument( "-u=");
    char *upw = ITK_ask_cli_argument( "-p=");
    char *ugp = ITK_ask_cli_argument( "-g=");
    initialize ( usr , upw , ugp );
    ITK_set_journalling(TRUE);
    export_bvr();
    ITK_exit_module(TRUE);
    return status;
}
/* Do BVR export */
static int export_bvr()
{
    int max_char_size = 80;
    char *xmlFileName;
    char *logFileName = (char *) MEM_alloc(max_char_size * sizeof(char));
    char   *itemid;
    char *rev;
    int    n_objects = 0;
    tag_t  *objects = NULL;
    // Create PIE session
    tag_t session;
    ERROR_CHECK( PIE_create_session(&session) );
    if ( (xmlFileName = ITK_ask_cli_argument( "-file=" )) == NULL )
    {
        display_help_message();
        return ( SampleError );
    }
    // Set the name of the XML file to export
    ERROR_CHECK( PIE_session_set_file(session, xmlFileName) );
    // Set the name of the log file
    sprintf(logFileName, "%s.log", xmlFileName);
    ERROR_CHECK( PIE_session_set_log_file(session, logFileName) );

    // get item id
    if ( (itemid = ITK_ask_cli_argument( "-item=" )) == NULL )
    {
        display_help_message();
        return ( SampleError );
    }
    // Get item revision
    if ( (rev = ITK_ask_cli_argument( "-rev=" )) == NULL )
    {
        display_help_message();
        return ( SampleError );
    }
    // Get tranfermode
    int n_transfer_modes;
    tag_t *transfer_modes;
    ERROR_CHECK( PIE_find_transfer_mode("ConfiguredDataExportDefault", "",
&n_transfer_modes,
```

```
     &transfer_modes) );
    if( n_transfer_modes == 0 )
    {
        printf("Error in finding default transfer mode\n");
        return(SampleError);
    }
    // Set the transfermode on the sessionion
    ERROR_CHECK( PIE_session_set_transfer_mode(session, transfer_modes[0]) );
    //To export the translations of a localizable properties, call
PIE__set_export_languages
    // with the language codes that you are interested in to set the locales to PIE
Session.
    // Get the topline of this item to export
    ERROR_CHECK( find_topline_of_itemid(itemid, rev, &n_objects, &objects) ) ;
    // Pass in the list of objects and do the export
    ERROR_CHECK( PIE_session_export_objects(session, n_objects, objects) );
    // Delete the session
    ERROR_CHECK( PIE_delete_session(session) );
    MEM_free(logFileName);
    return( ITK_ok );
}
/* Login */
static void initialize ( char *usr , char *upw , char *ugp)
{
    int status = ITK_ok;
    char *message;
    ITK_initialize_text_services( 0 );
    if ( ITK_ask_cli_argument( "-h" ) != 0  )
    {
        display_help_message();
        exit( EXIT_FAILURE );
    }
    status = ITK_init_module ( usr , upw , ugp );
    if (status != ITK_ok)
    {
        EMH_ask_error_text (status, &message);
        printf("Error with ITK_auto_login: \"%d\", \"%s\"\n", status, message);
        MEM_free(message);
        return ;
    }
    else
    {
        printf ("login to database successful.\n");
    }
}
/* Find topline for this item */
static int find_topline_of_itemid(char *itemid,char *rev, int *n_tags, tag_t **tags)
{
    tag_t   itemTag = NULLTAG;
    tag_t   rev_tag = NULLTAG;
    int local_num_tags = 0;
    tag_t *local_tags = NULL;
    // Find the required item
    ERROR_CHECK( ITEM_find_item (itemid, &itemTag) );
    if(itemTag == NULLTAG)
    {
        return(SampleError);
    }
    //Export only if rev is provided
    if (rev != NULL)
```

```
    {
        // Get item revision tag
        ERROR_CHECK( ITEM_find_rev(itemid,rev, &rev_tag) );
        tag_t     window = NULLTAG;
        tag_t     top_line = NULLTAG;
        // Create a window for BVR export of this item
        ERROR_CHECK( BOM_create_window(&window) );
        local_num_tags = 1;
        local_tags = (tag_t *) MEM_alloc(sizeof(tag_t) * 1);
        local_tags[0] = window;
        // Set this item as topline to export
        ERROR_CHECK( BOM_set_window_top_line(window,itemTag,rev_tag,NULLTAG,&top_line) );
        local_num_tags++;
        local_tags = (tag_t *) MEM_realloc (local_tags, sizeof(tag_t) * local_num_tags);
        local_tags[local_num_tags-1] = top_line;
    }
    *n_tags = local_num_tags;
    *tags = local_tags;
    return ITK_ok;
}
/* Displays help messgage */
static void display_help_message(void)
{
    printf( "\n\nsample_plmxml_itk_export:    PLMXML export of a simple BVR" );
    printf( "\n\nUSAGE:  sample_plmxml_itk_export -u=username -p=password -g=groupname
        -item=itemname -rev=itemrevision -file=filename");
    printf( "\n          -h    displays this message");
}
```

Customizing action rules

Actions are a set of methods that can be called before, during, and after the translation in a PLM XML session. Actions can also limit the translator from doing things, for example:

- Changing the ownership of the data on export.

- Modifying the check out status of data on export.

- Launching a utility after the import of data.

To register and use a custom action rule for PLM XML export:

1. Register your custom action handler by defining and assigning the **USER Register actions** extension in the Business Modeler IDE.

2. Once you register the custom action handler through the **PIE_register_user_action** function, the name that was used to register the action handler will appear in the **Action Handler** list in the PLM XML/TC XML Export Import Administration application.

Your main goal in writing a action rule is to create a C++ function for that rule. This function must adhere to the following constraints:

- The action rule must return **ITK_OK = int**.

- The action must be registered.

- Your function must use a parameter that is the tag of the session. The function signature must adhere to the following:

```
int your-function—name (METHOD_message_t* msg, va_list args)
```

- Your function must verify that the session tag is valid.

- Your function must perform ITK-based calls to the session using the public interface exposed to the session

Your function can then perform calls to the exposed objects from within the session.

The Teamcenter PLM XML/PIE framework allows for the pre, during, and post functional processing of action rules on behalf of the user. The typical functional processing that occurs for each of these types of action rules could be:

- **Pre-processing**

  Functions that need to verify or prepare the application, data, or Teamcenter to prepare for the translation. This rule is executed as follows:

  - **For export**

    Before the translation occurs.

  - **For import**

    Before the XML document is loaded.

- **During processing**

  If the translation engine verifies if you have a during action rule, then it is executed as follows:

  - **For export**

    After the translation and before saving the XML document.

  - **For import**

    After the document is loaded and before the translation.

- **Post-processing**

Post-processing is done after the translation is complete, but the session still has control over the translation.

The following code shows an example of ITK code for a pre-action rule. This example adds user information to the session prior to translation:

```
#include <itk/mem.h>
#include <tc/tc.h>
#include <tc/envelope.h>
#include <property/prop_errors.h>
#include <pie/pie.h>
#include <itk/bmf.h>
#include <tccore/custom.h>
#include <user_exits/user_exit_msg.h>
#include <tc/tc_macros.h>
#include <bmf/libuserext_exports.h>
extern USER_EXT_DLL_API int plmxml_user_exit_for_plmxml_actions
     (METHOD_message_t* msg, va_list args);
#include <bmf/libuserext_undef.h>
extern int custom_preaction_method(tag_t pie_session_tag)
{
    int    ifail     = ITK_ok;
    int max_char_size = 80;
    char *title = (char *) MEM_alloc(max_char_size * sizeof(char));
    char *value = (char *) MEM_alloc(max_char_size * sizeof(char));
    printf("Executing custom_preaction_method() ....\n");
    strcpy(title, "PLM XML custom pre-action session title");
    strcpy(value, "PLM XML custom pre-action session value");
    ITKCALL( PIE_session_add_info(pie_session_tag, 1, &title, &value) );
    MEM_free(title);
    MEM_free(value);
    return (ifail);
}
int plmxml_user_exit_for_plmxml_actions(METHOD_message_t* msg, va_list args)
{
    int    ifail = ITK_ok;
    ifail =  PIE_register_user_action( "customPreActionName" ,
                (PIE_user_action_func_t)my_custom_preaction_method );
    return ( ifail );
}
```

Customizing filter rules

Filter rules allow a finer level of control over the data that gets translated along with the primary objects by specifying that a user-written function is called to determine the operation applied against a given object. For example, you want the following to occur: if the item revision has a **UGMASTER** dataset, you want that translated and all other datasets skipped; if there is no **UGMASTER** dataset, you want to

export the JT dataset instead. This asks the closure rule to base its traversal on which child element is in existence, which the closure rule is not designed for. For this kind of control, you need a filter rule.

To register and use a custom filter rule for PLM XML:

1. Register your custom action handler by defining and assigning the **USER Filter Registrations** extension in the Business Modeler IDE.

2. Once you register the custom filter through the **PIE_register_user_action** function, the name that was used to register the filter rule appears in the **Filter Rule** list in the PLM XML/TC XML Export Import Administration application.

The following code shows an example of ITK code for a filter rule. This rule says if the object in the current call path is not released, do not process the object:

```
#include <itk/mem.h>
#include <tc/tc.h>
#include <tc/envelope.h>
#include <property/prop_errors.h>
#include <pie/pie.h>
#include <itk/bmf.h>
#include <tccore/custom.h>
#include <user_exits/user_exit_msg.h>
#include <tc/tc_macros.h>
#include <bmf/libuserext_exports.h>
extern USER_EXT_DLL_API int plmxml_user_exit_for_filter_actions
      (METHOD_message_t* msg, va_list args);
#include <bmf/libuserext_undef.h>
extern int plmxml_user_exit_for_filter_actions(METHOD_message_t* msg,
va_list args)
{
    int stat     = ITK_ok;
    printf("Executing plmxml_user_exit_for_filter_actions() ....\n");
    // register custom filter rule here
    ITKCALL( PIE_register_user_filter( "customFilterRuleName",
                (PIE_user_filter_func_t)custom_filter_rule_method ) );
    return stat;
}
extern int custom_filter_rule_method (void *msg)
{
    tag_t objTag             = NULLTAG;
    WSO_status_t* stats      = NULL;
    PIE_rule_type_t  result = PIE_travers_no_process;
    int   cnt       = 0;
    int   stat      = ITK_ok;
    int   released = 0;
    // Get the object from the call path
    ITKCALL( PIE_get_obj_from_callpath(msg, &objTag) );
```

```
        printf("Object tag = %d \n", objTag);
        // Get the oject status
        ITKCALL( WSOM_ask_status(objTag, &cnt, &stats));
        // usually only one status object returned, ignore others
        if ( (cnt >= 1) && (stats != NULL) )
        {
            POM_compare_dates (stats[0].date_released, NULLDATE, &released);
            if( released == TRUE )  // not released yet
            {
                result = PIE_travers_and_process;
            }
        }
        return result;
}
```

Setting the target path name

The **PIE_session_set_target_pathname** function helps to locate a PLM XML file in a directory other than the working directory. However, the file output does not recognize nor use this setting. If you are exporting bulk data files and your code looks like this example where the **PIE_session_set_target_pathname** function is called to set the path:

```
    PIE_session_set_transfer_mode(tSession, transfer_mode_tags[0]);
    PIE_session_set_revision_rule(tSession,tRevRule);
    PIE_session_set_file (tSession, "file-name.xml");
    PIE_session_set_target_pathname(tSession,"filepath\\");
    PIE_session_export_objects (tSession,1,&children[k]);
```

Change your code to remove the **PIE_session_set_target_pathname** function and add the file path to the file name as shown in this example:

```
    PIE_session_set_transfer_mode(tSession, transfer_mode_tags[0]);
    PIE_session_set_revision_rule(tSession,tRevRule);
    PIE_session_set_file (tSession, "filepath\\file-name.xml");
    PIE_session_export_objects (tSession,1,&children[k]);
```

If you are not exporting bulk data files, you can use the **PIE_session_set_target_pathname** function.

Data Exchange ITK

Exchanging data between Teamcenter and Teamcenter Enterprise

1. Site 1 requests objects from site 2.

2. Site 2 passes the objects through its scoper, which applies a transfer option set. The transfer option set contains a transfer mode with closure rules that defines the data to be passed along with the objects.

3.   The site 2 scoper then passes the data to its exporter, which exports the data in the native XML format of site 2 to the data mapper.

4.   The data mapper maps the data to the TC XML format and passes it to the site 1 importer.

5.   The site 1 importer receives the data, passes it through its scoper, and the data is available for site 1 use.

When you create ITK programs to execute an exchange, you must follow this process in the code.

There are several include files that contain the functions required for data exchange.

- **tie/tie.h**

  Contains functions needed for export and import of TC XML files: **TIE_export_objects**, **TIE_confirm_export**, and **TIE_import_objects**.

- **pie/pie.h**

  Contains functions needed for handling transfer option sets: **PIE_get_available_transfer_option_sets** and **PIE_describe_option_set**.

- **gms/gms.h**

  Contains functions that ensure that previously replicated data is synchronized between sites: **GMS_synchronize_site** and **GMS_identify_outdated_replica_objects**.

  It also contains functions that import and export objects even if Global Services is not enabled: **GMS_import_objects_from_offline_package** and **GMS_export_objects_to_offline_package**. These functions can also be used when Global Services is enabled.

Briefcase

Use Briefcase to exchange data with other Teamcenter sites, either online through a network connection or offline using an archive file. In addition to the **include** files used for Data Exchange between Teamcenter and Teamcenter Enterprise, there are additional functions required specifically for Briefcase:

- **publication/gms_publication.h**

  Contains functions required to move objects in and out of the archive file: **GMS_request_remote_export** and **GMS_request_remote_import**.

  All export and import calls use Global Services. The **GMS_request_remote_export** function requests that the selected object be exported to a given site. This function is used for both online and offline exporting. For offline exporting, set the **session_option_names** parameter to **offline** and its corresponding **session_option_value** parameter to **true**.

- **gms/gms_res.h**

  Contains the functions required to check files in and out: **GMS_RES _site_checkout**, **GMS_RES _site_checkin**, **GMS_RES _cancel_site_checkout**, and **GMS_RES _is_site_reserved**. Use these functions to lock the object at the remote site that you want to modify and unlock it when you are done.

TIE framework

A set of wrappers are provided that allow ITK programmers to extend the Teamcenter Import Export (TIE) framework. Following are the extension tasks you can perform:

- Register the serialize or de-serialize method.

  A typical use case for an ITK customization is when an installation has a user-defined object type extended from an existing POM type. TIE uses the extension rule functionality in Business Modeler Framework (BMF) to define the extension rule and handler for a type. ITK programmers can write C functions to define extension rules and handlers for their own types, and then populate the database with rules using Business Modeler IDE deploy. Wrappers are provided for ITK programmers to manipulate XML elements in a PLM XML document, such as get a PLM XML element by type name, create a PLM XML element by type name, add an attribute by name, and get an attribute by name.

  The message to define serialize method is **TIE_serialize**. The message to define the de-serialize method is **TIE_deserialize**. The input to both serialize and de-serialize methods is **TIE_message_t**.

- Register schema mapping.

  The mapping from Engineering Process Management data to XML elements is defined by property constants in the database and clauses in the property set. ITK programmers can use the Business Modeler IDE to add attributes of a custom class as required attributes in the database or add one clause for each attribute in the property set. No wrapper is needed.

- Manipulate the **TIE_message_t** structure.

  The **TIE_message_t** structure holds information about the current translation at any point of time during traversal. A full set of ITK functions is provided for the ITK programmer to get and set information in the **TIE_message_t** structure.

- Manipulate the **GenericSession** run-time object.

  The **GenericSession** object holds information given by the end user to perform a translation. A full set of ITK functions are provided for the ITK programmer to get and set information in the **GenericSession** object.

- Register session actions.

The ITK programmer can supply C functions to customize the import and export process. The following table shows when the action is executed:

Action	Export	Import
pre	Set after the exporter is ready for translation, and before the translation starts.	Set after XSLT is applied, and before the PLM XML file is opened for process.
during	Set after objects are exported, and before XSLT is applied.	Set after the PLM XML file is opened for process, and before translation starts.
post	Set after XSLT is applied.	Set after objects are imported.

The input to action is the **GenericSession** object. The wrap is provided for the ITK programmer to register actions.

- Register the ITK filters.

  The ITK programmer is able to supply C functions to examine each processed object based on object type. The input to filter is **TIE_message_t**.

The following is a list of the ITK functions implemented to register methods:

```
int TIE_register_user_filter
        (char*  filterRuleName, TIE_user_filter_func_t user_filter )

int TIE_register_user_action
        (char*  actionName, TIE_user_action_func_t user_action )
```

To use these ITK functions, include the **tie.h** file in C functions.

Multi-Site Collaboration ITK

Transferring site ownership

If your ITK program transfers Multi-Site site ownership over a network, it must call the **OBJIO_set_site_based_attribute** function to use the Synchronous Site Transfer (SST) protocol.

Additional information about this function can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Mapping application-specific IDs using Object Directory Services extensions

You can use the Application Reference (APPREF) ITK module to map application-specific unique IDs to a Teamcenter unique ID. The APPREF functions populate and query the **Publication Record** class managed by the Object Directory Services (ODS) with application-specific data. More than one application-specific ID can be associated with one Teamcenter item ID. The APPREF functions are called by the application that needs its IDs mapped to Teamcenter. For example, Teamcenter Integration for NX I-deas can call APPREF functions to map its GUIDs to a Teamcenter item ID or UID.

Additional information about the APPREF functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

The functions are called at ODS client sites which are sites that are configured to communicate with an ODS site using publication and remote search operations. The **TC_appref_registry_site** preference defines the ODS site that is used for all APPREF operations.

The following table shows the basic process and data flow for the APPREF module.

Step	ODS client site	ODS site
1		At startup, the ODS server registers the APPREF API handler.
2	Application calls the **APPREF** API.	
3	API calls the distributed ITK function.	The ODS server calls the registered APPREF API handler.
4		APPREF API handler performs the request and returns the results.
5	The distributed ITK function returns the results to the API.	
6	The API returns the results to the application.	

Legacy ITK considerations

Before Multi-Site Collaboration, when exporting an object it was possible to omit the destination or target site, which effectively produced an export file that could be imported by any site. With the implementation of Multi-Site Collaboration, you are now required to supply at least one target site.

Although this requirement has been addressed in the interactive **Export** dialog box and the **item_export** utility, it may cause some problems in existing user-written ITK programs that call OBJIO ITK functions to perform object export.

However, for those who want to continue using an existing ITK program without any change, you can do so by performing the following steps:

1. Relink your ITK program with the current libraries.

2. In the **Options** dialog box, accessed from the **Edit** menu in My Teamcenter, define the **TC_default_export_sites** preference with a list of the desired target sites. For example:

    ```
    TC_default_export_sites=
    SITE1
    SITE2
    ```

3. Define the **TC_USE_DEFAULT_EXPORT_SITES** environment variable and set it to **YES** as follows:

    ```
    export TC_USE_DEFAULT_EXPORT_SITES=YES
    ```

    Unless this is defined, the preference defined in step 2 is ignored.

4. Run the ITK program.

Caveats

The **TC_default_export_sites** preference is not used during interactive export operations.

This procedure is only applicable when not transferring site ownership. If the ITK program supports transfer of site ownership by calling the **OBJIO_set_site** function, this call is re-directed to call the **OBJIO_set_owning_site** function, thereby making the ITK program compliant with the Teamcenter requirements.

The **OBJIO_set_site** function is obsolete. Therefore, Siemens Digital Industries Software highly recommends that you modify any existing ITK programs to call the new **OBJIO_set_target_sites** (when not transferring site ownership) or **OBJIO_set_owning_site** (when transferring site ownership) functions.

USER_is_dataset_exportable user exit

The **USER_is_dataset_exportable** user exit helps refine Multi-Site dataset exporting. The normal Multi-Site framework allows you only to export all datasets with a particular relationship, but you cannot further sort out the datasets to export.

For example, assume you have this structure:

```
Item1
    ItemRevision1
        Dataset1 (attached to ItemRevision1 with the Reference relation)
        Dataset2 (attached to ItemRevision1 with the Reference relation)
```

If you attempt to export **Item1** to a remote site, you can select which relationships to be exported, and by selecting the **Reference** relation, you can export both **Dataset1** and **Dataset2**. However, you cannot export **Dataset1** by itself without also exporting **Dataset2**.

You can use the **USER_is_dataset_exportable** user exit to overcome this limitation. This user exit can be used to check if a dataset should be allowed to be exported based on its classification type. Following is the code for this user exit:

```
USER_EXITS_API int USER_is_dataset_exportable        (        tag_t
dataset_tag,
        int                 n_target_sites,
        tag_t *          target_sites,
        logical          is_transferring_ownership,
        logical          modified_objects_only,
        logical *    isExportable
    )
```

Parameter	Input or output	Description
dataset_tag	Input	Specifies the tag of the dataset object.
n_target_sites	Input	Specifies the number of target sites.
target_sites	Input	Defines the target sites list.
is_transferring_ownership	Input	Indicates if the Multi-Site transfer also transfers ownership.
modified_objects_only	Input	Indicates if Multi-Site transfer is transferring only modified objects or otherwise.
isExportable	Output	Indicates if this dataset tag should be Multi-Site exported or not.

Implement this user exit using standard user exit methods.

Additional information about the ITK **USER_is_dataset_exportable** user exit can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Customizing system administration modules

Customize Audit Manager

Perform the following steps to use ITK to define an action handler for Audit Manager:

1. Create a file (for example, **my_handler.c**) in the **\users_exits\code** directory with the following code:

```
#include <sub_mgr/subscription.h>
#include <user_exits/user_exits.h>
int PEC_Log_Handler(void* message, int nArgs, char** args)
{
        TcSubscriptionActionMsg_t* msg =
(TcSubscriptionActionMsg_t*)message;
                // add handler code here
        return 0;
}
```

2. Declare the function in the **user_exits.h** file:

```
extern USER_EXITS_API int PEC_Log_Handler(void* message, int nArgs,
char** args);
```

   TcSubscriptionActionMsg_t is defined in the **subscription.h** file.

3. Compile the file and build the **libuser_exits.dll** file.

4. Install the handler. For example:

```
%TC_BIN%\install_handlers –f=create –id=PEC_Log_Handler
–funcname=PEC_log_Handler –functype=1 –execmode=1  exectime=1800
```

5. In the Business Modeler IDE, add the new handler to the audit definition object you want to change and publish the change.

> Note:
>
> If your site is still using the legacy Audit Manager (Audit Manager solution 2, which is deprecated since Audit Manager solution 3 became available with Teamcenter 10.1), use the following step:
>
> • Modify the audit definition objects. You can modify the *TC_DATA\auditdefinition.dat* file to add the new handler to the audit definition object you want to change. Then either run the *TC_BIN***define_auditdefs -f=auditdefinition.dat** command or interactively log on to

the Audit Manager application in the rich client and modify the audit definition objects to add the new handler.

Customize Vendor management

Using Teamcenter vendor management, you can model vendor contributions to a product by capturing information about the vendors and the vendor parts they provide to satisfy the form, fit, and function of commercial parts.

You can customize your vendor management process with an ITK program. For example, you may want to write a program that does the following:

- Create a commercial part as a component of your product assembly.

  Use the **VMS_create_commercial_part** function.

- Create a bid package to provide to vendors so they can develop their quotes.

  Use the **VMS_create_bidpackage** and **VMS_create_bidpackage_lineitem** functions. You can also revise your bid package with bid package revisions, similar to items in Teamcenter.

- Create vendors that provide quotes.

  Use the **VMS_create_vendor** function. You can also create vendor roles, such as distributor or supplier, with the **VMS_create_vendor_role** function.

- Create vendor manufacturer parts and associated quotes that fulfill the requirements of your commercial parts. The manufacturer part and quote information is provided by your vendors. You can use different manufacturer parts for the commercial part in your product.

  Use the **VMS_create_manufacturer_part** function for the vendor part and connect it to your part information with the **VMS_add_mfg_part_to_comm_part** function. Use the **VMS_create_quote** function to create the quote and use the **VMS_add_quote** function to add the quote information to your bid package line item.

  > Note:
  >
  > In the user interface, manufacturing parts are called vendor parts.

Include the **vm\vms.h** file in your program and call the **VMS_init_module** function first to initialize the vendor management classes. For an example that uses the vendor management functions, open the *TC_ROOT***\sample\vms\vms_util.c** file.

Additional information about vendor management ITK functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Customize Volume Management

In Teamcenter, there are two methods to migrate infrequently used data to secondary and tertiary storage mediums: volume management (VM) and hierarchical storage management (HSM). Volume management is internal to Teamcenter. Hierarchical storage management can be used with third-party HSM software tools. In ITK, if you want to write code for VM or HSM policies, use the functions in the **sa/migration.h** include file.

If you want to use a third-party HSM software tool, Teamcenter provides a neutral API that calls the base method. Use the **HSM_migrate_files_msg** operation under the **HSM_Policy** business object in Business Modeler IDE. If you want a tight integration with the third-party tool, you can replace the base method with a customized method.

Additional information about the Volume Management functions can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Customize Report Builder

When you create report definitions templates, you can write ITK code to create report definitions and add your own customizations. In addition, there are functions that retrieve the existing report definitions based on different criteria, such as report type and source. To create a new report definition template, use the **CRF_create_report_definition** function. To retrieve existing report definitions based on different criteria, use the **CRF_get_report_definitions** function. To generate a report based on different input criteria, use the **CRF_generate_report** function.

The following example code shows how to create a summary report definition:

```
static void create_summary_report_defintion()
{
      tag_t     rd_tag= NULL_TAG;
      tag_t      qs_tag= NULL_TAG;
      char *rd_id = "offline_testing_api_ecube_2"; //OLD KEY IS
sum_rpt_for_items
      char *rd_name = "rd_create_testing";
      char *rd_desc = "rd_create_description";
      char *rd_source=NULL;
      char *queryName="Admin - Employee Information";
      tag_t query_t = NULLTAG;
        status_t stat = ImanQuery::find(queryName, &query_t);
```

```
            if ((stat != OK) || (query_t == NULLTAG))
            {
            printf("ERROR cannot find query %s s \n", queryName);
            return;
            }
            ITK_CALL( CRF_create_report_definition(rd_id,rd_name,rd_desc,NULL,0,
                                    NULL,0,rd_source,
                                    0,NULL,NULL,//params
                                    query_t, // queru source tag
                                    NULLTAG,
                                    0,
                                    NULLTAG,
                                    &rd_tag) );
        printf("New summary report rd Tag:  %d\n", rd_tag );
}
```

The following example code shows how to retrieve item report definitions:

```
static void list_reports()
{
  tag_t *tags = NULL;
  int n_tags=0;
  const char* itms="itm";
  ITK_CALL( CRF_get_report_definitions(itms,NULL,status,NULLTAG,&n_tags,
&tags) );
  printf("----BEGIN ITEM REPORTS ---n\n");
  for (int loopFound = 0; loopFound < n_tags ; loopFound++)
  {
        String uidString = Tag::tagToUid(tags[loopFound]);
        char *seeUid = uidString.smString();
        printf("  object tag is: %d object uid = %s\n",tags[loopFound],
seeUid);
        MEM_free(seeUid);
  }
}
```

The following example code shows how to find and generate a summary report:

```
static void find_and_generate_summary_report()
{
      tag_t      rd_tag= NULL_TAG;
      tag_t      qs_tag= NULL_TAG;
      const char* id="TC_2007_00_SUM_RPT_0004"; //4 for user stuff
      ITK_CALL( CRF_find_report_definition(id,&rd_tag) );
      printf("summary report rd Tag:  %d\n", rd_tag );
      int count=1;
      char **entryNames = (char **)MEM_alloc(sizeof(char *) * count);
      char **entryValues  = (char **)MEM_alloc(sizeof(char *) * count);
```

```
        int idx = 0;
        char  *report_path=NULL;
        char *param1= "User Id";
        char *value1= "*";
        entryNames[idx]=param1;
        entryValues[idx]=value1;
        tag_t dataset = NULLTAG;
        char *data_set_name=NULL;
        CRF_generate_report(rd_tag,dataset,0, NULL,
                            count,entryNames,entryValues, ///criteri stuff
                            data_set_name, /* dataset name */
                            &report_path );
        if ( report_path != NULL )
            printf(" Summary report generated at  %s",report_path);
}
```

Additional information about the Report Builder functions can be found in the *Integration Toolkit Function Reference*.
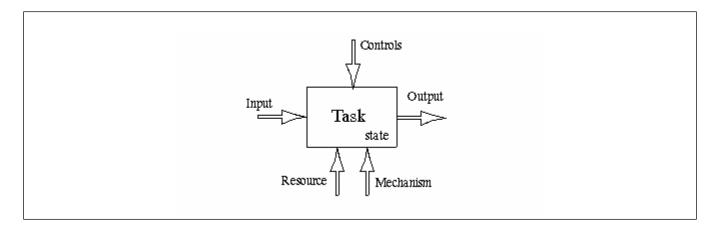
> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Workflow

Enterprise Process Modeling

The EPM module is a tool designed to facilitate the specification of processes (procedures), resources, and data constrained by a company's business rules. A typical product information management (PIM) system can be viewed as a process model consisting of procedures to be performed in a well defined sequence; a resource model which addresses the users, groups, roles and applications a business uses; and the data model which addresses the different types of data used to run the business's computer-aided applications. The company's business rules determine how the three models interact with one another. EPM addresses the procedures portion of this model.

A procedure can be viewed as a task or group of tasks which, when performed in a certain sequence with defined resources and rules, produces a defined output. The fundamental element of a procedure is a task. The following figure shows a model of this.

Task model

EPM is designed to support this model. EPM contains analogs to all the elements of the task model shown. The relationships are shown in the following table.

Task model	EPM
Controls	Business handlers
Mechanism	Action handlers
Input/output	Responsible party applications
State	State

A task is always in some state and it can change from one state to another by successfully performing an action. An action may have pre-conditions and business rules imposed on it. Any of the imposed business rules must be satisfied before the proposed action can be performed.

A business rule is a statement of some constraint on the performance of an action. It may be derived from empirical data and knowledge or other procedures. Tasks consume resources to produce outputs which, in EPM, are associated to the task by attachments. Each of the EPM elements is discussed in more detail in the next section.

EPM object model
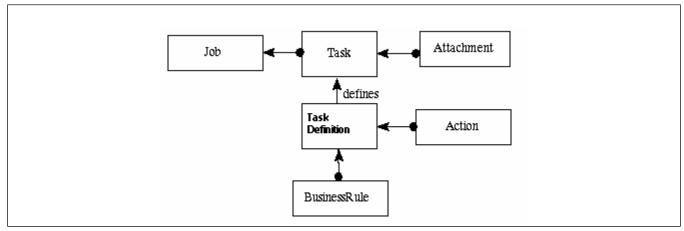
EPM object model components

EPM is comprised of the following components:

- Job

- Task

- Action

- Attachment

- Business rule

The following figure is a graphical representation of the EPM object model.

EPM object model

Jobs

A job is the run-time instantiation of a procedure. It is a collection of tasks just as a procedure is a collection of task definitions. A procedure is the system definition of a business process. Procedures are defined interactively through the **Procedures** dialog box.

Tasks

Introduction to task ITKs

A task is a rudimentary element in our process model. Data is associated to it and some action can be performed on it based upon a specific business rule. It is an instance of a task definition. Just as task definitions make up a procedure, tasks make up a job during run time. Subtasks can also make up a task to form an hierarchy of tasks.

You can use the **EPM_ask_sub_tasks** function to find the subtasks associated with a certain task. Similarly, you can use the **EPM_ask_sub_task** function to find one subtask attached to a task.

If you want to go up the hierarchy of tasks to find the parent of a given task, use the **EPM_ask_parent** function. Use the **EPM_ask_root_task** function to get to the top of the task hierarchy. To find out what tasks are assigned to a certain responsible party, use **EPM_ask_assigned_tasks**.

The **EPM_ask_description** function is also available to get a short description of a task in a character array. If you just want the name of the current task, use the **EPM_ask_name** function to get a character array containing the name of the task.

Success and failure paths

You can control condition task branching to follow success or failure paths by setting the value for the **result** attribute and then configure paths to accommodate all of the possible **result** values. By default, the **result** attribute is set to either **true** or **false**. However, you can set **result** to any string value and add as many valid values as you need. To do this, write and register a custom handler that uses the **EPM_set_task_result** function to set the **result** attribute to a string value. The code can contain any logic or algorithm you need.

You should configure the process template so there is a one-to-one relationship between the number of valid **result** values and the number of paths, unless you want to overload values onto a single path so the process traverses the same path for a number of results. For example, if you want five different valid values that can be used to set **result**, there must be five separate flow lines—one line for each of the five valid **result** values. Each line has a value associated with it, and this value must match one of the potential valid **result** values. If the value used to set the result does not match any of the path values, the process does not advance. By default, the **result** value is displayed as a label on the flow line associated with a workflow branch path. You can customized text labels by registering the text string.

Dynamic participants

If you use dynamic participants in your workflow and want to customize the participants, you can create participants with the **EPM_create_participants** function. To add or remove participants for an item revision, use the **ITEM_rev_add_participants** or **ITEM_rev_remove_participant** function, respectively.

Action handlers

Introduction to action handler ITKs

Action handlers are routines that act as the mechanism of the action to which they are associated. The association between an action and handlers is defined interactively in the **Task Definition** dialog box.

Actions can be either displayed during run time or triggered programmatically. To display the action string of the current action applied to a task during run time, use the **EPM_task_action_string** function. To trigger actions programmatically, use the **EPM_trigger_action** function to trigger an action in a specific task. This is useful in triggering a parallel task to start.

After writing your own handlers, you need to register them to the system using the **EPM_register_action_handler** function. This makes the handler available for use by the system administrator in implementing EPM procedures. The system administrator can enable the system to find your custom handler and execute it whenever the action with which the handler was associated is invoked on a task.

Predefined actions

EPM comes with predefined actions that can occur on any task.

Predefined action	Definition
Assign	Assigns a user or group to perform the work.
Start	Starts a task if it has not been started.
Complete	Marks a task as being completed.
Skip	For a given job, skips this subtask, but marks it complete for all other tasks that depend on it being completed before they can start.
Suspend	Suspends all work on this task until further notice.
Resume	Resumes the task to the state it was in before it was suspended.
Perform	Executes user-defined action handlers any time the user wants. The **Perform** action is executed based on a list of sub-actions that occur at various times throughout the task. It does not necessarily execute after the **Start** action but before the **Complete** action.

For example, the **Add Attachment** sub-action (**EPM_add_attachment_action**) executes a handler placed on the **Perform** action of the root task three times before executing a handler placed on the **Start** action.

The following are the available **Perform** sub-actions:

- **Add Attachment**

- **Remove Attachment**

- **Approve**

- **Reject**

- **Promote**

- **Demote**

- **Refuse**

- **Assign Approver**

- **Notify**

> Tip:
>
> Some **Perform** sub-actions can execute a handler multiple times. Before you place a handler on the **Perform** action, be sure that you want the handler to be executed more than once. For example, if

Predefined action	Definition
	the handler is **Notify** (**EPM-notify**), reviewers receive the same notification at each execution.

> Note:
>
> These do not change the state of the task.

State transitions

Some actions cause state transitions. For example, applying the **Assign** action to a task that is in the **Unassigned** state causes a state transition to the **Pending** state. Applying the **Start** action causes a state transition from **Pending** to **Started**. Applying the **Complete** action can cause a state transition from **Started** to **Completed**.

As the action handlers are executed, the task keeps track of its progress by setting an attribute called **state**. You can inquire about the state of a task using the **EPM_ask_state** function. Next use **EPM_ask_state_string** to get the string value of the state. Possible states that a task can take are as follows:

- **Unassigned**

- **Pending**

- **Started**

- **Completed**

- **Skipped**

- **Suspended**

State transition example

As an example of using state transitions, assume that our task is to do circuit analysis on a circuit design. During the circuit analysis procedure definition the rule and action handlers are associated with the **Start** action, the **Perform** action, and the **Complete** action of our task.

The **Start** action has rule handlers to check that all the files required for analysis are present. The rule handlers associated with the **Complete** action check to make sure the **Perform** action is invoked. The **Perform** action is where the actual analysis program is called by an associated action handler.

The initial state of the task is **Unassigned**. To start the task, initiate a job in the workspace using the circuit analysis procedure. The **Assign** action of the task is invoked automatically. If the assignment

action is successful, the task state transitions from **Unassigned** to **Pending** and the **Start** action is invoked. The business rule handlers associated with the **Start** action are then executed.

Assuming that the files required for analysis are all present, there is rule compliance with the **Start** action; therefore the task state transitions from **Pending** to **Started**. EPM invokes the **Complete** action automatically. Since the **Perform** action has not been invoked, there is no rule compliance in the **Complete** action; hence state transition from **Started** to **Complete** does not occur.

Next, invoke the **Perform** action and do the actual circuit analysis. Assuming the circuit analysis is successful, invoke the **Complete** action manually to finish the job. At this point there should be rule compliance to the **Complete** action business handler since the **Perform** action was done successfully prior to invoking the **Complete** action. The task state then transitions from **Started** to **Completed**.

Attachments

Attachments are other objects that are associated with the task during run time. These objects are typically items that need approval, supporting data, forms that need to be completed for the task, and so on. There are several built-in types of attachments.

A complete list of attachments can be found in the *Enterprise Process Modeling* section of the *Integration Toolkit Function Reference*. (The *Integration Toolkit Function Reference* is available on Support Center.)

Use the **EPM_ask_all_attachments** function to get an array of all attachments regardless of task type. To get attachments of a certain type, use the **EPM_ask_attachments** function, passing one of the valid attachment types as an argument. To add an attachment to a task, use the **EPM_add_attachments** function. You can remove an attachment with the **EPM_remove_attachments** function.

You can also define your own attachment types. This can be done by following a similar procedure for adding new actions. Modify the **tc_text_locale.xml** file to define the **EPM_attachment_***number* display name. The value of *number* must be in the range of 1000 to 32000 (1000 < *number* < 32000). For example:

```
<key id="EPM_attachment_1001">ECN</key>
```

For ease of maintenance, define the **ECN** macro in your local header file. For example:

```
#define ECN (EPM_user_attachment +1)
```

Business rule handlers

Business rule handlers are programs that check compliance to business rules before any action handlers can be executed within an action. Like action handlers, the association between action and rule handlers is defined interactively in the **Task Definition** dialog box. There are business rule handlers available in the system that you can readily use or you can write your own. Use the **EPM_register_rule_handler** function to register the rule handlers you have written.

Customizing a procedure with EPM

Introduction to customizing a procedure with EPM

A procedure can be customized with EPM in several ways. The simplest method is to associate available action or rule handlers from the toolkit to specific actions to obtain the desired result. With this method, you do not need to write a program or function to customize a procedure. If none of the available handlers meet your specific needs, then you can write a custom handler using the ITK. Such a custom handler can then be used along with the Teamcenter-provided handlers to specialize the behavior of a task.

A handler is a small ITK function designed to accomplish a specific job. There are two kinds of handlers: rule handlers and action handlers. Rule handlers enforce business rules. An action handler causes something to happen and usually is responsible for a state transition.

If the rule handler returns an **EPM_go**, then the action handlers are executed and a state transition occurs. If the rule does not return **EPM_go**, then the action handler does not execute and the state transition does not occur.

Creating action handlers

Writing your own action handler

Since action handler programs are essentially ITK functions, they should be written based on the same guidelines followed in ITK programs. The **epm.h** header file must be included in your source code. The standard interface for an action handler is as follows:

```
int sample_handler(EPM_action_message_t message)
```

The **EPM_action_message_t** is a typedef defined as:

```
typedef struct EPM_action_message_s
{
tag_t task;
EPM_action_t action;
EPM_state_t proposed_state;
TC_argument_list_t* arguments;
tag_t data;
} EPM_action_message_t;
```

The fields for **EPM_action_message_s** are as follows:

- **task**

  Task on which the action was triggered.

- **data**

System data.

- **action**

  Action that was triggered.

- **proposed_state**

  State that the task changes to after actions are completed without errors.

- **arguments**

  Argument specified in the procedure.

> Note:
>
> The action handler must return an int status code that can be compared to **ITK_ok** to determine if the handler was executed successfully.

To create your own action handler:

1. Write the action handler program following the standard interface and the ITK programming standards.

2. Compile your program using the compile script in the *TC_ROOT*/**sample** directory. The sample compile script writes the resulting object file in the current directory. Modify the compile script if the default behavior is not sufficient.

3. Register your new handler to the system by writing a module initialization function (or modifying it if one already exists) to call the **EPM_register_action_handler** function for the action handler you have written. Compile the module initialization function.

    > Note:
    >
    > Complete step 4 only if this is the first handler you are registering. You can skip this step for the subsequent handlers that you are going to write.
    >
    > You can also register handlers using the Business Modeler IDE.

4. Modify the **user_gsshell.c** file in the *TC_ROOT*/**sample** directory to add a call to your module initialization function in the **USER_gs_shell_init_module** function. Compile the **user_gsshell.c** file using the *TC_ROOT*/**sample/compile** script.

5. Create the user exits library with the *TC_ROOT*/**sample/link_user_exits** script. This script creates the **libuser_exits** sharable library from all of the object files in your current directory.

Action handler example

The following is an example of an action handler:

```
int EPM_system(EPM_action_message_t msg)
{
int err = ITK_ok;
char* arg = 0;
if(TC_number_of_arguments(msg.arguments)>0)
{/* get first argument */
arg = TC_next_argument(msg.arguments);
system(arg);
}
return err; /*return an ITK error code if an error/
occurred*/
}
```

The first call to the **TC_next_argument** function gets the first argument, given the **Arguments** field from the structure, specified in the task definition windows for this handler. Each successive call to **TC_next_argument** gets the next argument on the list. Use the **TC_init_argument_list** function to start over at the beginning of the list and get the first argument. All action handlers should return either **ITK_ok** if no error occurred or the ITK error for the error that did occur.

Creating rule handlers

Writing your own rule handler

The standard interface for a business rule handler is:

```
EPM_decision_t sample_rule_handler/
(EPM_rule_message_t message)
```

To write your own rule handler:

1. Write the rule handler program following the standard interface and the ITK programming standards.

2. Compile your program using the compile script in the *TC_ROOT*/**sample** directory. The sample compile script writes the resulting object file in the current directory. Modify the compile script if the default behavior is not sufficient.

3. Register your new handler to the system by writing a module initialization function (or modifying it if one already exists) to call the **EPM_register_action_handler** function for the rule handler you have written. Compile the module initialization function.

EPM_decision_t

> Note:
>
> Complete step 4 only if this is the first handler you are registering. You can skip this step for the subsequent handlers that you are going to write.

4. Modify the **user_gsshell.c** file in the *TC_ROOT*/**sample** directory to add a call to your module initialization function in the **USER_gs_shell_init_module** function. Compile the **user_gsshell.c** file using the *TC_ROOT*/**sample/compile** script.

5. Create the user exits library with the *TC_ROOT*/**sample/link_user_exits** script. This script creates the **libuser_exits** sharable library from all of the object files in your current directory.

EPM_decision_t

The **EPM_decision_t** is a user-defined type defined as:

```
typedef enum EPM_decision_e
{
EPM_nogo,
EPM_undecided,
EPM_go
} EPM_decision_t;
```

The fields for **EPM_decision_t** are as follows:

- **EPM_nogo**

  Constraints for rule or rule handler are not satisfied; action and state transition should not occur

- **EPM_undecided**

  Still unknown whether the decision is a go or no go because of insufficient or pending data; action and state transitions should not occur

- **EPM_go**

  Rule or handler passed; constraints satisfied

EPM_rule_message_t

The **EPM_rule_message_t** is a typedef defined as:

```
typedef struct EPM_rule_message_s
{
tag_t task;
EPM_action_t proposed_action;
```

```
TC_argument_list_t* arguments;
tag_t tag;
} EPM_rule_message_t;
```

The fields for **EPM_rule_message_t** are as follows:

- **task**

  Task on which the action was triggered

- **proposed_action**

  Action that was triggered

- **arguments**

  Argument specified in the procedure

- **tag**

  System data

Rule handler example

The following is an example of a rule handler:

```
EPM_decision_t
EPM_check_responsible_party(EPM_rule_message_t msg)
{
int s;
EPM_decision_t decision = EPM_nogo;
char * userName;
tag_t aUserTag, responsibleParty;
s=EPM_ask_responsible_party(msg.task, &responsibleParty);
if (responsibleParty == NULLTAG)
{
decision = EPM_go;
}
if (s == ITK_ok && decision == EPM_nogo)
{
s = POM_get_user (&userName, &aUserTag);
MEM_free (userName);
if (s == ITK_ok)
{
if (aUserTag == responsibleParty)
decision = EPM_go;
}
```

```
    }
    return decision
    }
```

Registering rule handlers

All rule handlers must be registered. The following example illustrates how you register a rule handler.

> Note:
>
> You can also register handlers using the Business Modeler IDE.

The following example uses the **USER_epm_init_module** function that registers a function called **EPM_check_responsible_party** through the **EPM_register_rule_handler** function as the **EPM-check-responsible-party** handler.

When defining tasks interactively, use the registered name of the **EPM-check-responsible-party** handler to add the handler to an action instead of the full name of the function, **EPM_check_responsible_party**:

```
int USER_epm_init_module()
{
int s;
s = EPM_register_rule_handler("EPM-check-responsible-party", "",
EPM_check_responsible_party);
return s;
}
Register the USER_epm_init_module() function through
USER_gs_shell_init_module().
The USER_gs_shell_init_module() function exists in user_gsshell.c located
in the
user_exits directory. The registration is shown as follows:
USER_gs_shell_init_module()
{
...
USER_epm_init_module();
}
```

Compile all of the above mentioned code and create a user exits library. Information about creating a user exits library can be found in the *Integration Toolkit Function Reference*.

> Note:
>
> The *Integration Toolkit Function Reference* is available on Support Center.

Validation rules

Using validation rules

A validation rule is an instance of a **Validation Data** object with the following additional run-time attributes:

- A validation agent that can be an NX CheckMate checker.

- The dataset types where the validation agent is applicable.

- Whether the check is mandatory or optional. A mandatory check must both run and pass. An optional check must run but does not have to pass.

- Event names where the check is applicable. When applicable event names are specified for a validation agent (checker), the check result is verified only when these events happen. If no event name is specified, the checker applies to all events. The event name can contain an asterisk as a wildcard. Event names can be marked as exclusive.

- A list of parameter and value pairs. The parameter values need to be verified from the validation result log file.

Validation rules are defined based on your business model. In a production environment, you can define multiple validation rule files to suit different business processes.

Validation rule file

Validation rules can be defined in an XML file called a validation rule file. A validation rule file defines all validation rules for a validation rule set. Follow these guidelines when you create a validation rule file:

- The **Rule_Set** tag attributes **name**, **description**, **user**, and **date** are mandatory. These attributes are mapped as Teamcenter validation rule set object attributes when the XML rule set is imported.

- The **Rule** tag defines a rule.

  - **Checker tag**

    This tag is mandatory. Its **name** attribute is the checker class name, not the display name.

  - **Mandated tag**

    This tag is mandatory. If a check is mandatory (the check must run and pass), type **yes** between the opening and closing tag. If not, type **no**. When a check is not mandatory, the check must run, but the result status is ignored.

  - **Datasets tag**

This tag is mandatory. You must define the list of applicable dataset types.

- **Events tag**

  This tag is optional. The events contain the event values where the check is applicable. If you do not define an event value list, then the rule applies at all times. The event values can contain an asterisk as a wildcard. The values listed can be exclusive when the **exclude** attribute is set to **yes**.

  If you define the **class** attribute for the **Events** tag, you set which target revision attribute value should be tested against event values list.

- **Parameter tag**

  Only parameters that logged in profile level (for a profile check result log file) are supported. Child checker parameters must be promoted into the profile level to be verified by the validation rule APIs.

- You can define rules in any order in the validation rule file.

- The same checker can be instanced in the definition of more than one rule in the same rule file.

The following sample shows an example of a validation rule file:

```
<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="validation_rule.xsl"?>

<Rule_Set
          name="Test_rule_set_1"
          description="Power Train Event 0 at Milestone 3"
          user="your_user_id" date="2005-12-25"
          xmlns="http://www.plmxml.org/Schemas/
PLMXMLSchemaValidationRule"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.plmxml.org/Schemas/
          PLMXMLSchemaValidationRule validation_rule.xsd"    >

<Rule>
    <Checker name="mqc_check_dim_attach_solid"></Checker>
    <Mandated>no</Mandated>
    <Datasets>
       <type>UGMASTER</type>
       <type>UGPART</type>
    </Datasets>
    <Events class="ITEMREVISION:owning_group" exclude="yes">
       <value>chassis.pe.company</value>
       <value>powertrain.pe.company</value>
    </Events>
```

```
    </Rule>

    <Rule>
        <Checker name="mqc_examine_geometry"></Checker>
        <Mandated>yes</Mandated>
        <Datasets>
            <type>UGPART</type>
            <type>UGMASTER</type>
        </Datasets>

    </Rule>

    <Rule>
        <Checker name="mqc_check_dim_attach_solid"></Checker>
        <Mandated>no</Mandated>
        <Datasets>
            <type>UGPART</type>
        </Datasets>

    </Rule>

    <Rule>
        <Checker name="mqc_exam_geom_combo"></Checker>
        <Mandated>no</Mandated>
        <Datasets>
            <type>UGMASTER</type>
            <type>UGPART</type>
            <type>UGALTREP</type>
        </Datasets>
        <Events exclude="no">
            <value>Status*3</value>
            <value>Status_4</value>
        </Events>
        <Parameters>
            <Parameter operation="=" title="Check Tiny objects"
                value="TRUE></Parameter>
            <Parameter operation="=" title="Check Misaligned Objects"
                value="TRUE"></Parameter>
        </Parameters>

    </Rule>

    <Rule>
        <Checker name="mqc_examine_geometry"></Checker>
        <Mandated>no</Mandated>
        <Datasets>
            <type>UGMASTER</type>
            <type>UGPART</type>
```

```
        </Datasets>
        <Events exclude="no">
           <value>S*</value>
        </Events>
        <Parameters>
           <Parameter operation="=" title="Save Log in Part"
              value="TRUE></Parameter>
        </Parameters>

    </Rule>

    </Rule_Set>
```

Validation rule APIs

The following sample shows how to call validation rule APIs to perform a validation result check with the validation rules:

```
validation_rule_t *validation_rules = NULL;
    int             num_rule = 0;
    candidate_validation_result_t *candidates = NULL;
    int             num_candidates = 0;
    /*******************************************
    * Get Validation Rule list
    *******************************************/
    ifail =
    VALIDATIONRULE_get_rules (
               rule_item_revision,
               NULL,
           current_event,
               &validation_rules,
               &num_rule
               );
    /*******************************************
  * Before calling VALIDATIONRULE_get_candidate_results() to get
candidate
  * result list, user may want to put codes here to resolve all
unresolved
  * rules.
    *******************************************/
    ifail =
    VALIDATIONRULE_get_candidate_results(
               target_objects,
               count,
               validation_rules,
               num_rule,
               &candidates,
               &num_candidates
               );
```

```
/*********************************************
* check validation result for each candidate
*/
for( int inx=0; inx<num_candidates; inx++)
{
    ifail = VALIDATIONRULE_verify_result(
            &(candidates[inx])
            );
}
// Before exit, free up allocated memories
VALIDATIONRULE_free_rules(validation_rules, num_rule);
VALIDATIONRULE_free_results(candidates,num_candidates);
```

EPM hints

Perform action handlers

The rule and action handlers associated with the **Perform** action get called every time an action with an action code greater than or equal to **EPM_perform_action** is triggered. Clicking the **Perform** button calls the **EPM_trigger_action** function with the action equal to **EPM_perform_action**.

Many existing ITK functions also call the **EPM_trigger_action** function to allow you to customize Teamcenter. The following table lists actions that are triggered inside each ITK function.

ITK function	Action code
EPM_add_attachment	EPM_add_attachment_action
EPM_remove_attachment	EPM_remove_attachment_action
CR_set_decision	EPM_approve_action
CR_set_decision	EPM_reject_action
CR_promote_target_objects	EPM_promote_action
CR_demote_target_objects	EPM_demote_action

Branching in your handler

You can branch logically in your handler by using **message.action** in an **if/else** block. For example:

```
if (message.action == EPM_add_attachment_action)
        {
                /* do something */
        }
        else
        {
```

```
                              /* do nothing */
            }
```

Avoiding infinite loops in perform actions

Infinite loops can occur when performing actions if logic is not included to test the values of the actions being triggered. For example, the following action handler, **my_action_handler**, is executed when a **Perform** action is triggered:

```
int my_action_handler(EPM_action_message_t msg)
{
.........
.........
err = EPM_add_attachment (...............................);
.........
.........
return err;
}
```

When the **err = EPM_add_attachment (..........)** statement is executed, **my_action_handler** is called again as a part of the **EPM_add_attachment** call. This is because **EPM_add_attachment** calls **EPM_trigger_action** with an action code of **EPM_add_attachment_action**. Since all action codes greater than **EPM_perform_action** get sent to all of the perform action handlers, this causes an infinite loop.

To avoid this scenario, you should check the action being triggered inside each handler. To avoid entering an endless loop in the example shown above, add the following lines to your code:

```
if(msg.action > EPM_perform_action)
return ITK_ok;
```

For example:

```
int my_action_handler(EPM_action_message_t msg)
{
if(msg.action > EPM_perform_action)
   return ITK_ok;
......... /* initial lines of code for action */
......... /* handler initiation */
err = EPM_add_attachment (...............................);
......... /* final lines */
......... /* of code */
return err;
}
```

Note:

Infinite loops can occur when **EPM_trigger_action** handler with the task action **EPM_complete_action** is placed on the **Assign**, **Start**, or **Complete** of a task.

A. How to develop an application

Process for developing an application

Use the Business Modeler IDE to develop your own application. Use the Business Modeler IDE for:

- Business model modifications

- Service-oriented architecture (SOA) services and operations creation

- Teamcenter server code development

The Business Modeler IDE provides all these capabilities in a seamless manner.

For a walkthrough of how to use the Business Modeler IDE to develop an application, use the following example:

- Business object: **Comment**

  - Capture additional information on any **WorkspaceObject**

  - Properties: **relatedObject**, **text**

  - Operation: **copyToObject**

- Service: **getCommentsForObjects**

Follow this process:
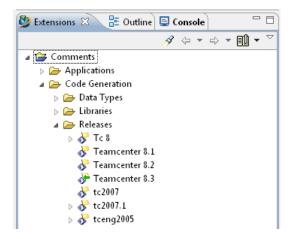
1. **Define business objects** to represent objects to work with in the application.

2. **Define operations** to be placed on the business objects.

3. **Define services** to expose the business logic in the enterprise tier to clients.

4. **Generate stubs** to be used for implementing your code.

5. **Implement your code**.

6. **Build your code**.

7. **Package and deploy** your application.

Develop an application: define business objects

Define business objects to represent items to work with in the application.
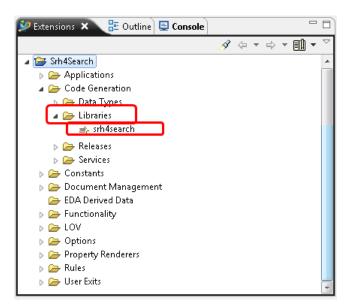
1. Set an active **release**.

    Active release helps in associating operations to a release and deprecating and later removing of an operation when it is not needed anymore.
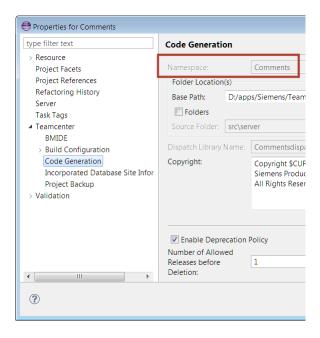
2. **Create a library** for code and set the library as active.

    Business object interfaces that contain the operations get added to the library.

3. Set the code generation namespace.
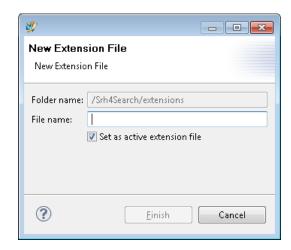
Each template can specify a namespace This namespace is associated with C++ classes generated for business objects and for data types created in that template. The namespace protects against collision of C++ class names from different templates (for example, **Teamcenter::Item** and **Comments::Item**).

4. Create extension files.

Extension files enable proper organization of Business Modeler IDE work. To create an extension file:
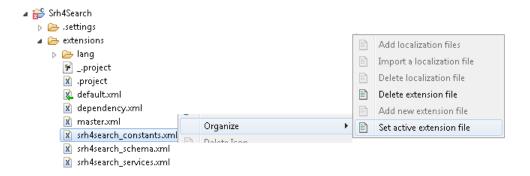
a. In the **Project Files** folder, select the extension directory under the project.

b. Right-click the **extensions** folder and choose **Organize→Add new extension file**.

c. Type the desired extension name.

Following are some example of extension files you can create to organize your work.
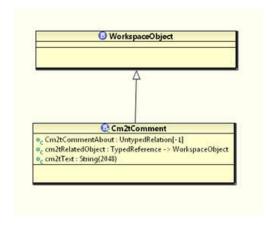
Extension file name	Contains
*project-name*__business_object_constants.xml__	Business object constants
*project-name*__deepcopy_rules.xml__	Deep copy rules
*project-name*__display_rules.xml__	Business object display rules
*project-name*__extension_attach.xml__	Extension attachments
*project-name*__extension_rules.xml__	Extension rules
*project-name*__global_constants.xml__	Global constants
*project-name*__grm_rules.xml__	GRM rules
*project-name*__lovs.xml__	Lists of values (LOVs)
*project-name*__property_constants.xml__	Property constants
*project-name*__rule.xml__	Rules
*project-name*__schema.xml__	Business objects
*project-name*__services.xml__	Service-oriented architecture (SOA) operations

5. Before proceeding with a task, activate the proper extension file:

    a. Right-click the project where you want to save your work and choose **Organize→Set active extension file**.

    b. Open the **Project Files\extensions** folders and select the active extension file, for example, **default.xml**. A green arrow symbol appears on the active extension file.
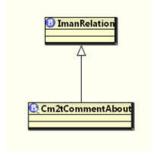
6.   Define a business object.

a.   Create a business object.

For example, define a **Comment** business object as a child of the **Workspace** business object.

b.   Add required properties.

c.   Define a relation.
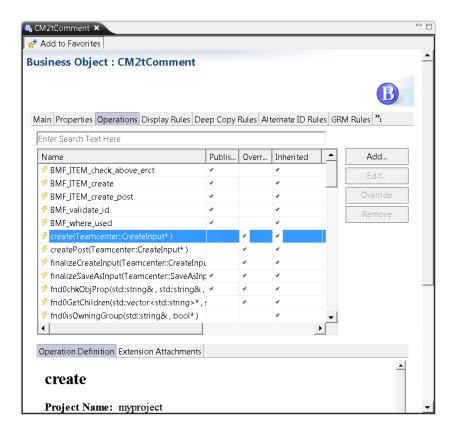
d.   Define GRM rules.
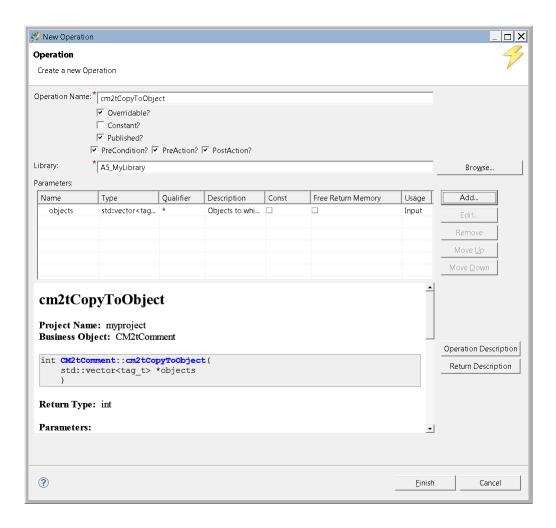
e. Define deep copy rules.

Develop an application: define operations
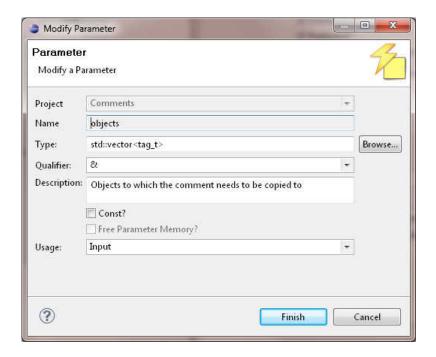
Operations are functions or methods An operation is inherited to child business objects An operation can be overridden at the child business object.

1. **Create operations**.

Add a new operation with parameters. As you enter operation information, the code preview is updated.
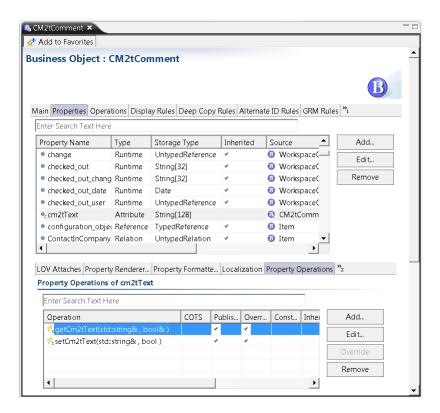
Do the following for operations:

- Specify parameters.

- Ensure the return value is **int**.

- Set status of operation.

- Enable preconditions, preactions, and postactions.

- Make overridable.

- Publish the operation.

2. **Create property operations**.

   Property operations are generic operations to set and get a property on a business object. You can configure the safe getter and setter operations on custom properties.
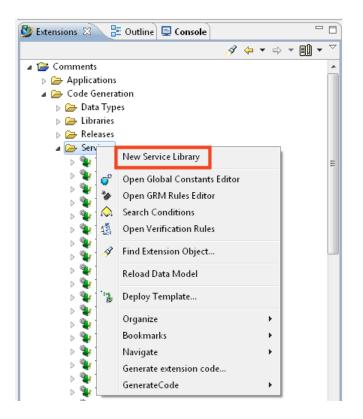
   For example, in the following figure, the **getCm2tText** getter operation is used for the **Cm2tText** property.
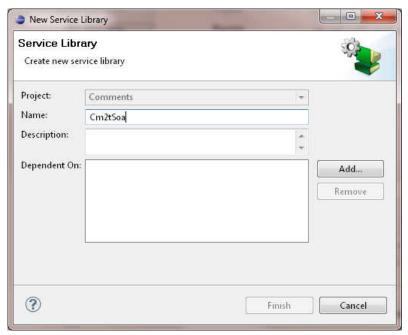
Develop an application: define services
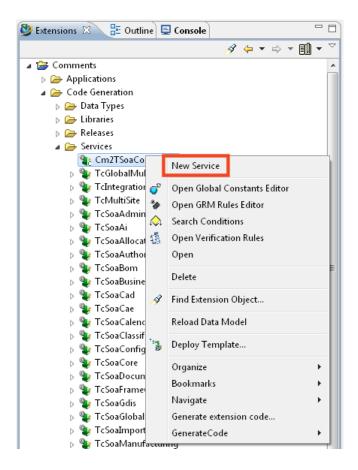
Services expose the business logic in the enterprise tier to clients.

1.   **Create a service library** to host the services.
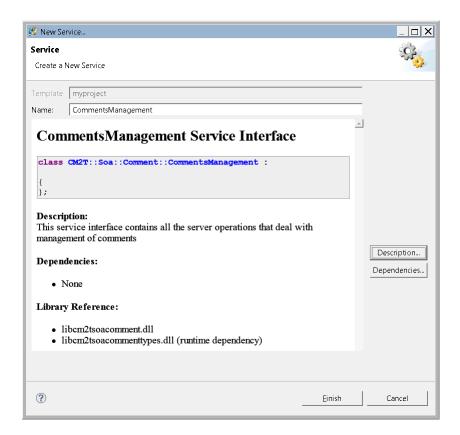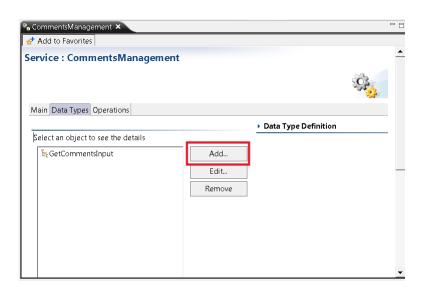
2. **Create services** in the library.
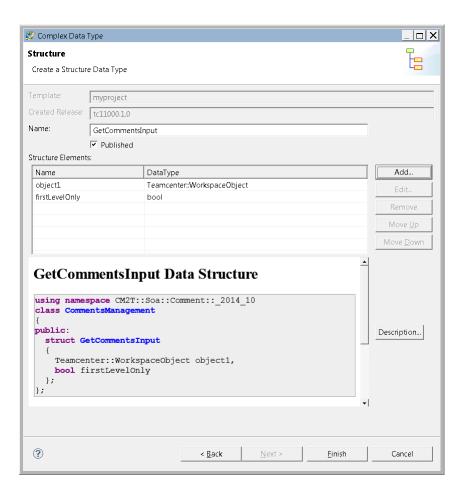
Each service library may contain one or more service interfaces. Each service interface contains one or more service operations and data types used by those operations.
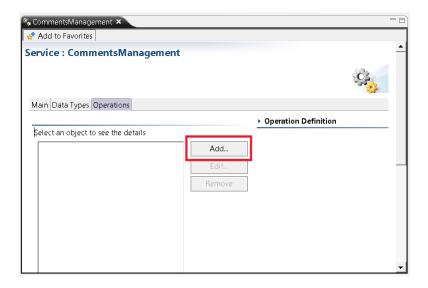
3. Create **data types**.
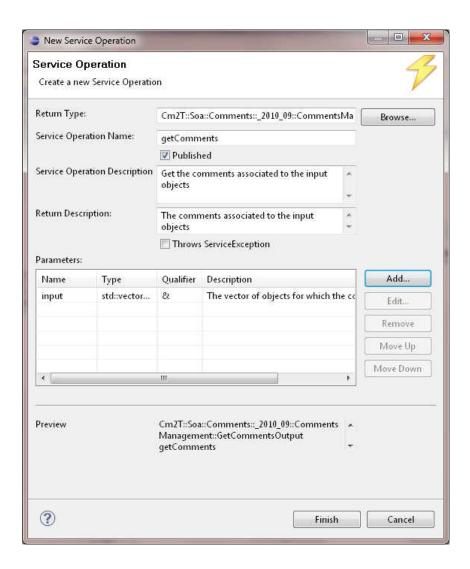
Service operations require input and output. Create data types to represent the input and output. Data types must be defined before defining the operations. Common data types are provided.

4. **Create a service operation**.
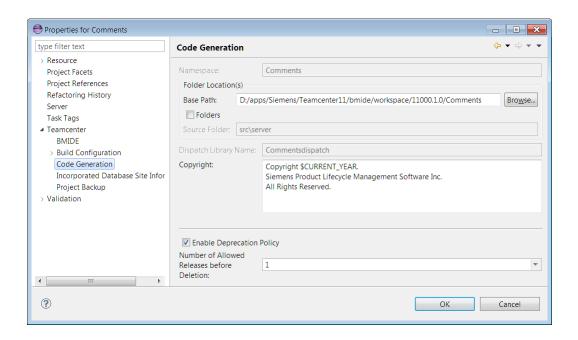
Clients call the service operations to execute the necessary business logic. Service operations can identify operations as published or internal. Making them set-based helps reduce client server communication when needed.

Develop an application: generate code
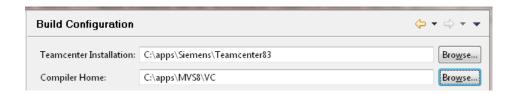
After you create operations, you must generate code.

1. Configure code generation.
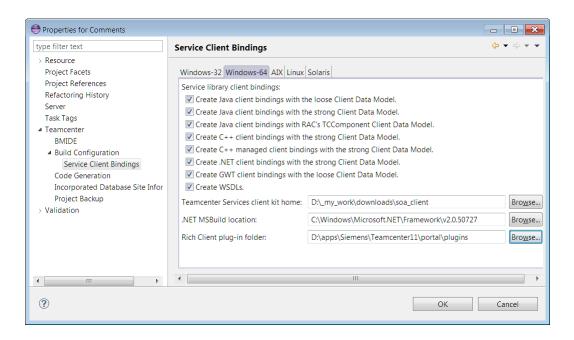
When you set code generation properties for your template, set the following:

- Locations for C++ code

- Namespace for C++ classes

- Copyright to be placed in generated code

- C++ compiler location

- Teamcenter installation location
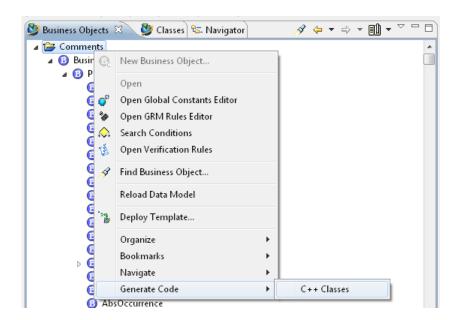
2. Configure **services stubs generation**.

   a. Right-click the project and choose **Properties**.

   b. Choose **Teamcenter→Build Configuration→Service Bindings** in the left pane.

   c. Unzip the **soa_client.zip** file (from the same location as the **tem.bat** file) to a directory. Type this location in the **Teamcenter Services client kit home** box.

   d. Select the required bindings. (It depends on the client code needed.)
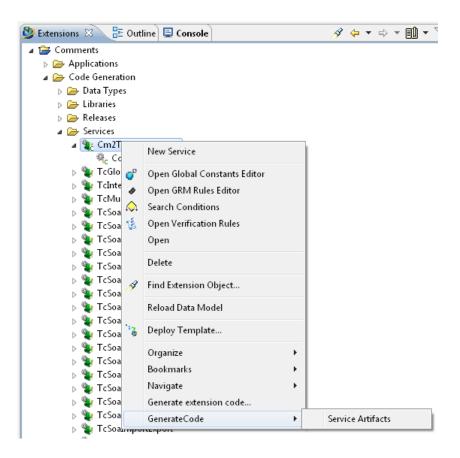
3.   Generate code.

Generate code after business object operations and services are defined. Regenerate if the definition changes. This generates interfaces and implementation stubs for business objects and generates the transport layer and implementation stubs for services.

The following example shows generating C++ code.

The following example shows generating services code.

Develop an application: implement code

After you generate stubs, write **implementation code**.
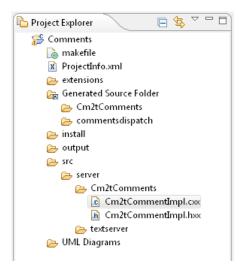
1. Open the C/C++ perspective.

    Code development is done in the C/C++ perspective. The CDT (C/C++ Development Tools) plug-in is an Eclipse-based C++ integrated development environment.

2. **Implement business logic**.

    An outline of the operation is created by the code generation step. The implementation file is found in the **Project Explorer** under the library for the business object.

Add the business logic for the operation.

3. **Implement service operation**.

An outline of the service operation is created by the code generation step. The service interface implementation file is found in the **Project Explorer** under the service library.
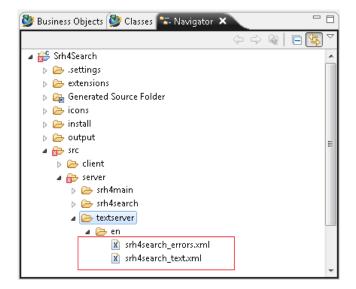
Add the business logic for the operation.

```cpp
#include <commentsmanagement1009impl.hxx>

using namespace Cm2T::Soa::CommentsManagement::_2010_09;
using namespace Teamcenter::Soa::Server;

CommentsManagementImpl::GetCommentsOutput CommentsManagementImpl::getCommentsForObject (
        const GetCommentsInput& input )
{
    // TODO implement operation
}
```

4. Create text server files.

   Text server files enable storage and retrieval of localized and nonlocalized information.

   Translatable resources are stored in an XML-formatted files. Error messages are stored in ***_errors.xml** files. Other translatable resources are stored in ***_text_locale.xml** files. Both types of files are stored in a language-specific directory. Nontranslatable resources are stored in ***_text.xml** files in a language-neutral directory (**no_translation**).

   The Business Modeler IDE creates the resource files in the *project***/src/server/textserver/en** directory.

a. Create error messages.

Error base defines the code area. The codes are multiples of 1000 added to the **error_base xml** attribute, for example:

```
<errors module="ss" error_base="5000">
```

Each error is defined with an **id** XML attribute:

```
<error id="64">The removal of the definition has failed for the following
preference:
%1$.error id="64">The removal of the definition has failed for the following
preference: %1$.>
```

The error number is equal to **error_base** + *id* (for example, **5064**) and must be given an associated C++ symbol, for example:

```
#define PREF_cannot_delete_preference_definition   (EMH_SS_error_base + 64)
```

The same code area symbols are usually defined in the same file.

A. Define the error base in the **emh_const.h** file to avoid conflicts:

```
#define EMH_SS_error_base   5000
```

B. Choose your error base within the **919000-920000** range. Replace the required information in the line:

```
<errors module="<enter the solution name>" error_base="<enter error base
number>">
```

Declare errors in this file, ensuring that error numbers not exceed **920000**.

---

    b.   Create other resources.

        A.   Create a **no_translation** directory at *project***/src/server/textserver/**.

        B.   Add a *project***_text.xml** file to the directory.

        C.   Add the following line to the *project***_text.xml** file:

```
<xi:include href="project_text_locale.xml" parse="xml"/>
```

        D.   Add the nontranslatable resources to the *project***_text.xml** file.

        E.   Copy the *project***_text.xml** file to *project***_text_locale.xml** and add translatable resources to the *project***_text_locale.xml** file. Use unique key identifiers in both files.

    c.   Provide the translation for all supported languages. Ensure packaging of the files.

    d.   Deploy text server files.

    Add missing information for upgrade. This ensures proper deployment.

    Edit the *project***/install/feature_***project***.xml** file. Add a section for the upgrade (based on the section for install):

```
<install>
    <code>
        <textserver file="emh_text.xml" include="${feature_name}_errors.xml"/>
        <textserver file="tc_text.xml" include="${feature_name}_text.xml"/>
    </code>
</install>
<upgrade>
    <code>
        <textserver file="emh_text.xml" include="${feature_name}_errors.xml"/>
        <textserver file="tc_text.xml" include="${feature_name}_text.xml"/>
    </code>
</upgrade>
```

5.   Implement utilities.

Utilities are C/C++ programs using ITK calls. They can be used to achieve very lengthy tasks in one call. Sometimes utilities are used during installation and upgrade scripts. Utilities must be run on the Teamcenter server host. They can also be used for testing C and C++ code.

Use Business Modeler IDE to implement the utility. Ensure packaging of the compiled executable if needed by the installer or upgrader.
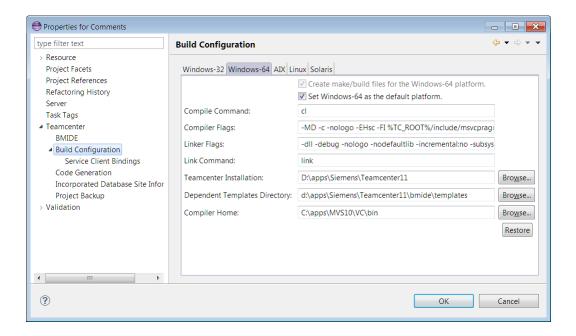
Develop an application: build libraries

After writing implementations, build the libraries containing server code.
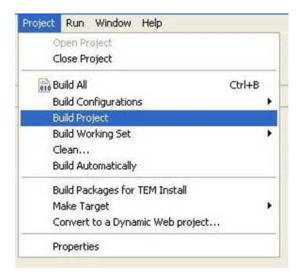
1.   Set up the build configuration.

     Right-click the project, choose **Properties**, and choose **Teamcenter**→**Build Configuration**.
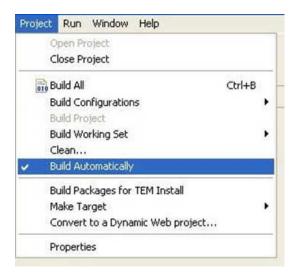
     Set up the Teamcenter installation location, the C++ compiler location, the compiler command and flags, and the link command and flags.
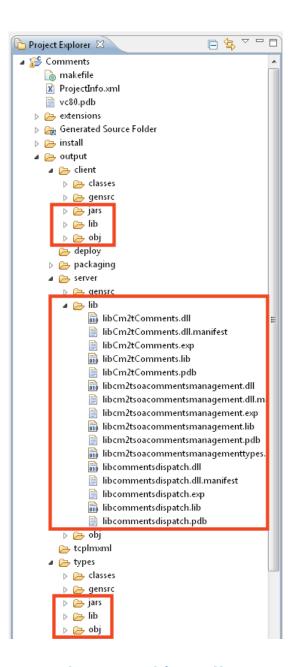

2.   To build code, on the menu bar, choose **Project**→**Build Project**.

To turn on automatic building, choose **Project→Build Automatically**. The build occurs automatically when changes to the code are saved.

Build files are saved to the appropriate output locations.

Develop an application: package and install

When all files are built, generate a distribution package for installation.

1. To package, choose **BMIDE→Generate Software Package**, or choose
   File→New→Other→Business Modeler IDE→Generate Software Package.

   Package files include the following:

   • Metamodel definitions

   • Built libraries

- Enterprise tier libraries

- Web tier files

- SOA client libraries (that are consumed in the clients)

- Installation and upgrade scripts and data files

2. Install the packaged template files using Teamcenter Environment Manager (TEM).

    The packaged template can be deployed through TEM or Deployment Center to a production database.