Cúram 8.1.2

Business Object Module Development Guide

Note

Before using this information and the product it supports, read the information in Notices on page 49

Edition

This edition applies to Cúram 8.1, 8.1.1, and 8.1.2.

Contents

1 Developing Business Object Modules for Configuration Transport Manager

Use this information to learn how to transport business objects between systems. A business object type is a logical grouping of administrative data that defines and governs a particular set of functions. A business object module is a piece of code responsible for performing the bespoke processing that is required to transport instances a particular business object type.

1.1 Overview

This document provides details of the development activities that are necessary in order to enable the transport of administrative Business Objects from one system to another via CTM.

The document is intended to be used by any development community that wants to enable the transport of the administrative Business Objects that they have developed between different application systems using Configuration Transport Manager (CTM).

Prerequisites

The document assumes that the reader is familiar with the following components.

See the following guides for more details:

-
-
-
- Google Guice 2

Terminology

This section defines some of the key terms that are used throughout the document.

Business Object Type

A Business Object Type is a logical grouping of administrative data that defines and governs a particular set of functionality. Each Business Object Type consists of the collection of data (that is, entities) that are required to configure the system to use and/or act on the functionality that it represents.

For example, the set of administrative data related to a *Benefit Product* involves grouping the entities related to *Benefit Product* such as *Product*, *ProductDeliveryPattern*, *ProductCategory*, etc. Therefore, *Benefit Product* is a Business Object Type. For further examples, .

Business Object

A Business Object is an instance of a Business Object Type.

For example, it's possible that there are multiple *Benefit Product* configurations available on the system. Each such configuration is a Business Object.

Business Object Modules

A Business Object Module (BOM) is a piece of code responsible for performing the bespoke processing that is required in order to transport instances a particular Business Object Type. Several types of BOM must be implemented for every transportable Business Object Type, with each type of BOM being responsible for a different part of the flow involved in transporting the Business Object.

1.2 BOM Overview

This section provides an overview of the responsibilities of the various BOM types, describes the supporting infrastructure available to assist in providing the required functionality, and summarizes the other activities involved in making a Business Object Type transportable.

The core CTM Infrastructure is responsible for executing the transport operations that are common to all Business Object Types. It co-ordinates the overall flow involved in the transport of a Business Object, delegating to other components where necessary. In particular, the CTM infrastructure delegates to the BOMs for the specific Business Object Types that are being transported at points in the flow where Business Object – specific activities must be performed. For example, when an XML document containing the content of a particular Business Object is required, the CTM infrastructure will invoke on the BOM responsible for producing the XML document for the Business Object.

Implementing BOMs

In concrete terms, developing a BOM involves providing an implementation of the BOM interfaces appropriate to the Business Object Type that is being made transportable. There are in total eleven different types of BOM that may need to be implemented for each Business Object type. However, note that it is not generally necessary to provide implementations of all of the BOM types. Out-of-the-box implementations are provided for five of the BOM types, and, provided these are suitable, bespoke BOMs do not need to be provided for these.

The different BOM types are illustrated in the following table:

S.No	Interface	Responsibility	OOTB Implementation Available
1	AuthorisationBOM	Determine whether or not the user is authorized to act on a Business Object.	Y – OOTB implementation uses SecurityBOM to determine authorisation.
2	DeleteBOM	Delete a Business Object.	N
3	DependentBOM	Provide a list of other Business Objects upon which a Business Object is dependent.	N
4	ExistenceBOM	Determine whether or not there is a Business Object already present on the target system.	N
5	InformationalBOM	Provide various information about the Business Object	N
6	PostCommitActionBOM	Perform a Business Object - specific activity after the transaction applying a Change Set has been committed.	Y – No-Op OOTB implementation provided.

S.No	Interface	Responsibility	OOTB Implementation Available
7	PreCommitActionBOM	Perform a Business Object - specific activity immediately before the transaction applying a Change Set is committed.	Y - No-Op OOTB implementation provided.
8	PreCommitAction TypeBOM	Perform a Business Object Type - specific activity immediately before the transaction applying a Change Set is committed.	Y - No-Op OOTB implementation provided.
9	RevertChangeSetConstruction HandlerBOM	Add extra Business Objects to a Change Set being created for revert purposes.	Y - No-Op OOTB implementation provided.
10	ReadAndUpsertBOM	Create an XML document with the content of a Business Object; Populate the database with the content of the XML document.	N
11	SecurityBOM	Provide the SIDs that a user is required to have in order to read and write the Business Object.	N – but if an AuthorisationBOM is provided, then a SecurityBOM does not need to be implemented.

Details on how the BOMs are used are provided in the next section (CTM Core Process Flow on page 11), which describes the CTM Core Process Flow. Additionally, for more detailed technical information on each of the BOM types, please refer to the Javadoc of the interfaces, which are all contained in the *curam.util.ctm.bom package*. Finally, further details on implementing the BOM interfaces are provided in the next chapter, 1.3 Developing BOMs on page 14

CTM Core Process Flow

To illustrate where the BOM Infrastructure APIs are used and invoked, the two core CTM flows which involve BOMs are now described. These are the *Release* operation and the *Apply* operation.

The Release Operation

The *Release* operation captures and freezes the content of the Business Objects contained in a Change Set.

The operation starts by performing a check to see whether or not a user is permitted to read the relevant Business Objects. It then collects all of the Business Object contents and converts them into XML fragments. It gathers the fragments into a single Change Set XML document, and then saves the Change Set XML document to a release area. All of these activities take place in a single transaction.

As part of the operation, BOMs are used as follows:

- *AuthorisationBOM* (if provided) or *SecurityBOM* : Check that the user has the appropriate permissions to read each Business Object in the Change Set
- *ReadAndUpsertBOM* : Read the Business Object contents from the data store, and convert to an XML document.

Apply Operation

The *Apply* operation provides the functionality to make a Released Change Set "live".

The actions that occur during the *Apply* operation broadly fall into three categories:

- Pre Apply Phase
- Apply Phase
- Post Apply Phase

Both the Pre-Apply Phase and the Apply-Phase take place in the same transaction. The Post-Apply phase takes place in a separate transaction, after the Pre-Apply / Apply transaction has been committed. The phases are now described in more detail.

Pre Apply Phase

The first step of the Pre-Apply phase is to validate the content of the Change Set to see if the Change Set is eligible to be applied. As the Apply operation involves both database read and write operations, the user performing the operation must have the appropriate read and write permissions for each Business Object defined in the Change Set. If the user does not have the appropriate permissions, then the *Apply* process is terminated.

Next, the infrastructure creates a *revert Change Set* for undo purposes. It does so by capturing the current state of the database with respect to each of the Business Objects in the Change Set. That is, for each Business Object defined in the Change Set, the infrastructure will identify if the Business Object already exists in the target database. Since the business logic for determining the existence of a Business Object is very specific to the Business Object type, the infrastructure delegates the call to the *ExistenceBOM* in order to get the desired results.

As part of this phase, BOMs are used as follows:

- *AuthorisationBOM* (if provided) or *SecurityBOM* : Check that the user has the appropriate permissions to read and write each Business Object in the Change Set
- *ExistenceBOM* : Determine whether or not an instance of each Business Object already exists on the target system.

Apply Phase

Once the Pre-Apply phase has successfully completed, processing proceeds to the Apply Phase.

In this phase, the infrastructure will iterate over each Business Object and then invoke either an *upsert* or *delete* operation, depending on whether the Business Object is to be *upserted* or deleted[1]. To perform these operations, either the *ReadAndUpsertBOM* or *DeleteBOM* are invoked as appropriate. After all Business Objects in the Change Set have been *upserted* or *deleted*, the *PreCommitActionBOM* for each Business Object is invoked. This is in order to perform any pre-commit activities that are required for the Business Object. Following this, the *PreCommitActionTypeBOM* is invoked for every Business Object Type which has at least one Business Object instance in the Change Set.

As part of this phase, BOMs are used as follows:

- *ReadAndUpsertBOM* : Add or update the Business Object in the target system database.
- *DeleteBOM* : Delete the Business Object from the target system database.
- *PreCommitActionBOM* : Perform any pre-commit actions for the Business Object.
- *PreCommitActionTypeBOM* : Perform any pre-commit actions for the Business Object Type.

[1] Note that deletion of Business Objects is currently only supported in the revert Change Sets that are automatically created for Undo purposes

Post-Apply Phase

Once the Apply Phase has successfully completed, the transaction will be committed and the post apply phase will be executed.

The post-apply phase involves invoking on the *PostCommitActionBOM* for each Business Object in the Change Set. This BOM can perform any activities that are required after a Change Set has been committed.

As part of this phase, BOMs are used as follows

- *PostCommitActionBOM* : Perform any post-commit actions for the Business Object.

BOM Infrastructure

A set of infrastructural classes, known as the BOM Infrastructure, are provided to assist in implementing BOMs. This infrastructure provides default implementations for some of the most common operations.

The main classes provided are as follows:

AbstractEntityBOBuilder

Classes known as Entity Business Object Builders need to be implemented for the entities in a Business Object Type. These produce XML fragments for the entity and also perform the low-level CRUD actions involved in *upserting* the entity's content.

The *AbstractEntityBOBuilder* class provides capabilities that are helpful in implementing the *ReadAndUpsertBOM*. Essentially, it provides two pieces of functionality:

- It is used to build entity instance information from XML fragments
- It acts as a wrapper, hiding the low level CRUD operations for an entity.

For more details, please refer the javadoc for *curam.ctm.bom.util.impl.AbstractEntityBOBuilder*. Additionally, more detail on using this class is provided in the 1.3 Developing BOMs on page 14.

Abstract BOM

The *AbstractBOM* class provides functionality for a number of the BOM types. A Business Object Module can extend from *AbstractBOM* to gain access to this common functionality.

For further information, please refer the javadoc for *curam.ctm.bom.util.impl.AbstractBOM*. Additionally, more detail on using this class is provided in the 1.3 Developing BOMs on page 14.

The Range Aware Key Server

Another important piece of infrastructure used in integrating Business Object Types with CTM is the Range Aware Key Server (RAKS). The standard Key Server generates keys that are unique within a particular system, but which may be duplicated across different systems. The RAKS is a new Key Server implementation that is responsible for providing identifiers (for example, primary keys) that are unique across the set of systems that form a System Landscape (that is, the set of systems between which Business Objects may be transported).

Entities that are part of transportable Business Objects must use primary keys generated by the RAKS instead of the standard Key Server. This is to ensure that there are no key clashes when a Business Object is transported and applied on a target system. It's worth noting that entities that already use the standard Key Server can be changed to use the RAKS.

Runtime Data

An important point to note is that Business Object Types should not contain entities that consist of runtime data. To avoid this scenario, entities should be designed to either contain runtime data or administrative (configuration) data. The former (runtime entities) should not be transported via CTM, and so should use primary keys generated by the standard Key Server; that is, they should not use primary keys generated by the RAKS. The latter (administrative entities) can be transported, and so must use primary keys generated by the RAKS.

1.3 Developing BOMs

This section describes how to develop the BOMs for a Business Object Type from scratch. To illustrate the process of developing a BOM, this chapter uses an artificial example application that manages some pieces of configuration information. Using this example application, the various steps involved in analyzing the configuration entities are explained. Then, the content of the Business Object Types is determined. Next, the BOMs for the Business Object Types identified are implemented. Finally, the steps involved in testing the transport of the Business Objects are explored.

The Example Application

Let us assume that there is a user interface for an application that mimics a traditional Personal Information Management (PIM) application. Also, let us assume that this functionality is available in the *pim* component. Usually, a PIM application provides facilities to manage personal information about a user. Since managing personal information is quite complex and what we will be discussing here is just for illustration purposes, we will not be covering the complex details that are involved in standard PIM application.

Also let us assume that this application provides the following higher level functionality:

- The application manages personal information about a *user*. This involves creating, editing and destroying user information.
- The application also supports managing related personal information such as *to-do* and *note* items.
- It is possible from the application to link *user* to multiple *to-do* and *note* items.
- Furthermore, while creating *notes*, it is possible to associate them with particular *folders*.
- *Folders* can be separately managed and it is possible to assign multiple *permissions* to a *folder*.

Imagine that the application has two separate screens to manage the above configuration information:

- Folder Screen
- User Screen

Folder Screen

The following actions can be performed via the Folder Screen:

- Managing *folder* information – e.g. creation, modification and removal of folders.
- Managing *permissions* related to a *folder* – e.g. adding and removing the permissions associated with a particular *folder*.

Let us imagine that the equivalent home page for *folder* is developed and available in a UIM file named *Folder_home.uim*. This page accepts a mandatory page parameter named *folderID*, whose value is used to identify and show the relevant folder information in the home page.

Entities

For managing *folder* and its related *permission* functionalities, let us assume that the following entities are involved:

- *Folder*
- *FolderPermission*
- *FolderPermissionLink*

The following diagram represents the entity relationship model for the folder entities through UML.

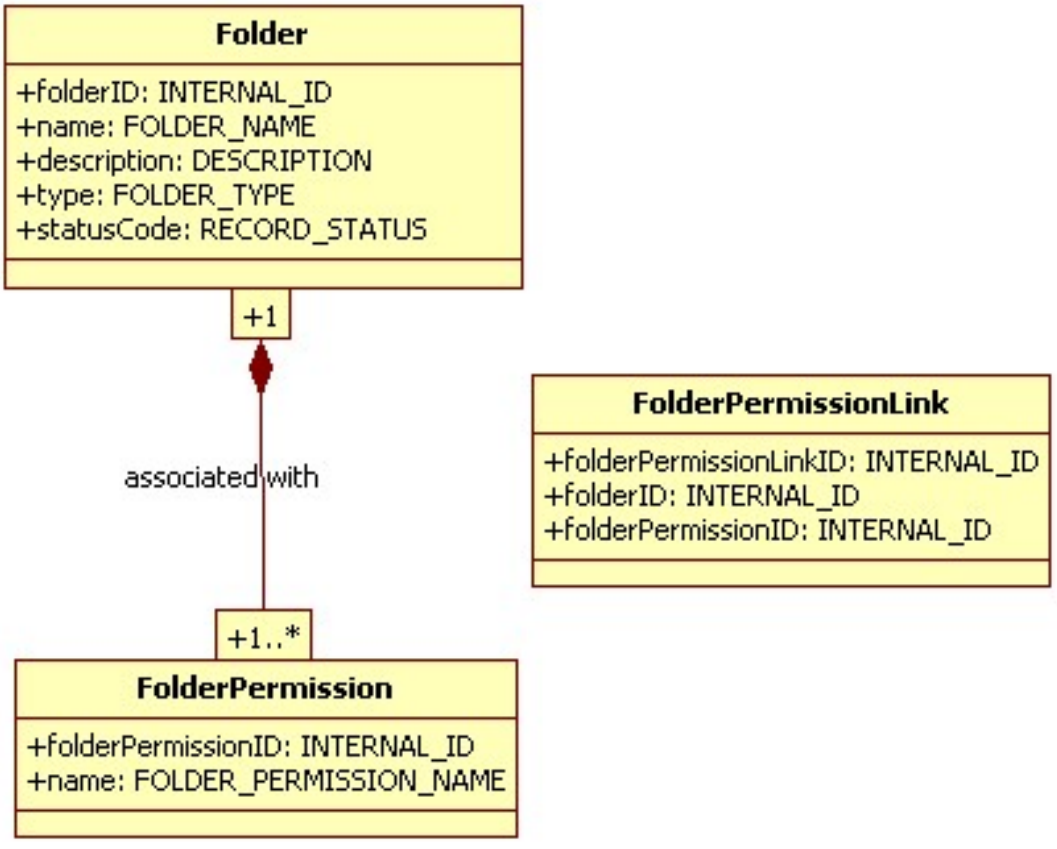

Figure 1: UML representation for folder screen entities

The following sections describe the low level details of these entities – e.g. the set of attributes, the code table associations, and the primary and foreign key details.

Folder

The entity *Folder* represents a standard *folder* object. The table below lists the various attributes that make up the *Folder* object. Note that the attributes *type* and *statusCode* have code table relationships to *FolderType* and *RecordStatus* respectively.

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
folderID	Y		
name			
description			
type			FolderType
statusCode			RecordStatus

FolderPermission

The entity *FolderPermission* represents a permission object that can be assigned to a *folder*. For simplicity, other than the primary key attribute, this entity has only one attribute, *name*, which stores the name of the permission. Also note that this attribute has a soft relationship with the code table *FolderPermissionName*.

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
folderPermissionID	Y		
name			FolderPermission Name

FolderPermissionLink

Since it is possible for a *folder* to have multiple *permissions* associated with it, the association between a *folder* and its *permissions* are captured in this entity.

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
folderPermissionLinkID	Y		
folderID		Folder.folderID	
folderPermissionID		FolderPermission. folderPermissionID	

Code Tables

The Folder screen entities have dependencies on code tables.

This section lists the set of dependent code tables:

Code Table Name	Code	Description
FolderType	FT_PR	Private
	FT_PU	Public
FolderPermissionName	FPN_FC	Full Control

Code Table Name	Code	Description
	FPN_READ	Read
	FPN_WRITE	Write

User Screen

The User Screen provides functionality for the following tasks:

- Managing *user* information – e.g. creation, modification and removal of *users*.
- Managing *to-do* and *note* items for *users* - i.e. creation, modification and removal of *to-do* and *note* items.
- Associating *to-do* and *note* items with *users*.
- Assigning *notes* to *folder* objects.

Let us imagine that the equivalent home page for *user* is developed and available in the UIM file *User_home.uim*. This page accepts a mandatory page parameter by the name *userID*, whose value is used to identify and show the relevant user information in the home page

Entities

The following entities are used to manage *user* and the related functionality:

- *User*
- *ToDo*
- *UserToDoLink*
- *Note*
- *UserNoteLink*
- *Category*

The following diagram represents the entity relationship model for the user entities through UML.

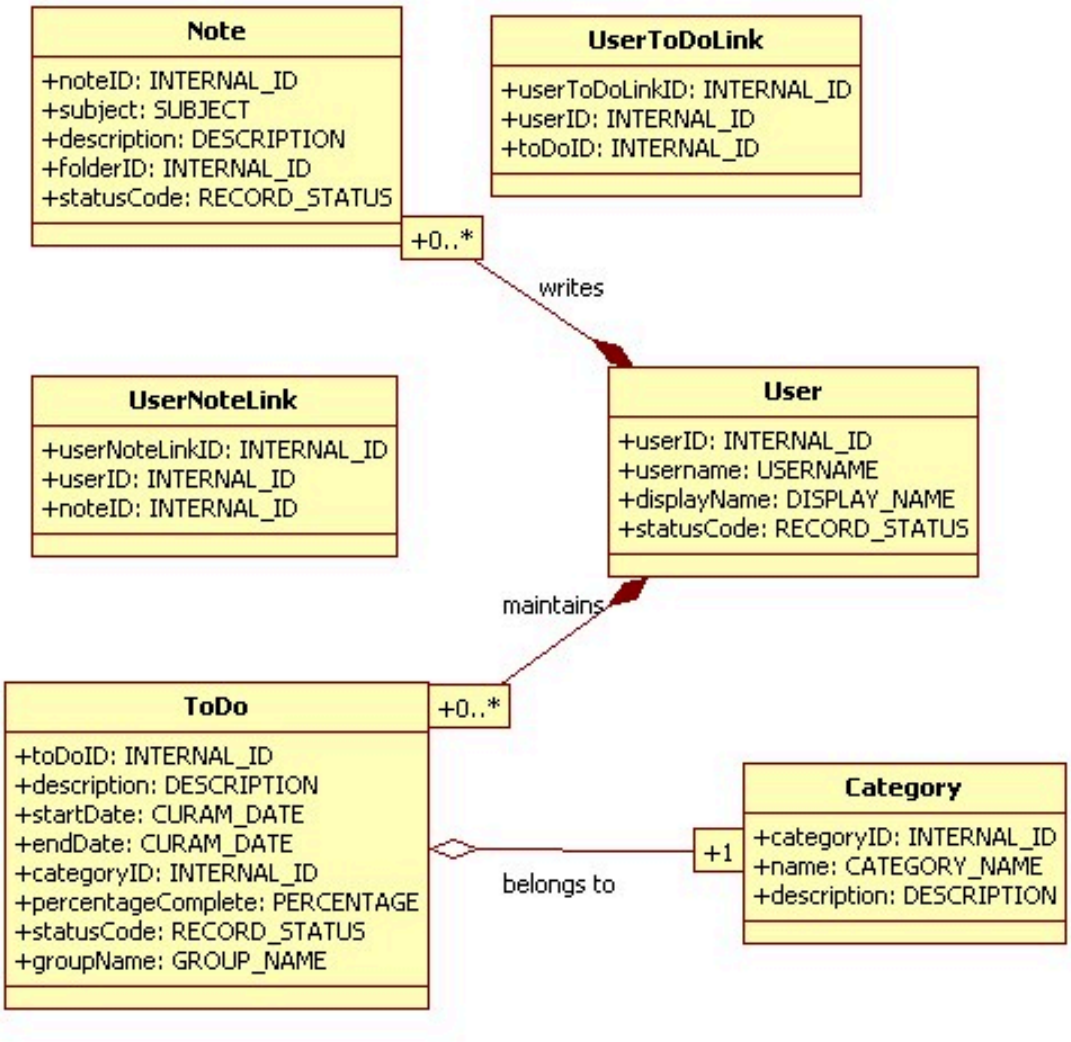

Figure 2: UML representation for user screen entities

The following sections describe the low level details of these entities – e.g. the set of attributes, the code table associations, and the primary and foreign key details.

User

The *User* entity represents the user whose personal information is being managed by the application. The following table provides further details about this entity:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
userID	Y		
username			

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
displayName			
statusCode			RecordStatus

ToDo

The *ToDo* entity represents the information that forms a *to-do*. For simplicity, this entity contains the following information:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
toDoID	Y		
description			
startDate			
endDate			
categoryId		Category. categoryID	
percentageComplete			
statusCode			RecordStatus
groupName			GroupName

Note that the attribute *categoryID* has a foreign key relationship with the entity *Category*. Also, note that the columns *statusCode* and *groupName* have associations with the *RecordStatus* and *GroupName* code tables respectively.

UserToDoLink

This link entity provides association between *User* and *ToDo* entities. Since there exists a one-to-many relationship between a *user* and *to-do* items, the association between them is captured in this entity:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
userToDoLinkID	Y		
userID		User.userID	
toDoID		ToDo.toDoID	

The *Note* entity represents the standard note that contains information such as *subject*, *description*, etc. The table below provides more information about this entity:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
noteID	Y		

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
subject			
description			
folderID		Folder.folderID	
statusCode			RecordStatus

Also, it is possible to associate a *note* with a *folder*. This is implemented via a foreign key relationship with the *Folder* entity, using the *folderID* attribute.

UserNoteLink

Since it is possible for a *user* to have many linked *note* items, the *UserNoteLink* entity is introduced in order to maintain an association between the *User* and *Note* entities:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
userNoteLinkID	Y		
userID		User.userID	
noteID		Note.noteID	

Category

The *ToDo*, entity has a foreign key relationship with the *Category* entity through the attribute *categoryID*. The following table lists the information available on the *Category* entity:

Column Name	Primary Key Column	Foreign Key Details	Code Table Association
categoryID	Y		
name			CategoryName
displayName			
description			

Code Tables

The User screen entities have dependencies on code tables.

This section lists these code tables:

Code Table Name	Code	Description
CategoryName	CN_BIZ	Business
	CN_PERS	Personal
	CN_HOLI	Holiday

Code Table Name	Code	Description
GroupName	GN_PRIV	Private
	GN_SHARED	Shared

BOM Development Methodology

This section describes the general steps involved in implementing BOMs.

In broad terms, this involves the following activities:

- Analyzing Business Object Types
- Implementing BOMs
- Testing BOMs

Analyzing Business Object Types

As part of analysis, the set of configuration entities have to be identified and then grouped into logically separate Business Object Types.

This process involves the following steps:

- Identifying the Configuration Entities
- Group entities into Business Object Types
- Define Business Object Identifiers
- Ensure that the Configuration Entities use RAKS generated identifiers

Identifying the Configuration Entities

The aim of CTM is to transport configuration data. Therefore, the development group must be able to identify all of the entities that constitute configuration information in an application. Because CTM is not designed to support runtime data, caution has to be exercised in order to ensure that the configuration entities don't have dependencies upon runtime entities. That is, as part of the analysis, the configuration entities have to be checked to ensure that there are no foreign key constraints or any form of soft dependent relationship on runtime entities. If any such cases are encountered, then the recommendation is to refactor the entity design so that separate entities are used for configuration and runtime purposes.

That is, one set of entities to hold only configuration data and the other set of entities to hold the runtime data. Note that the runtime entities can depend on configuration data, but the reverse is not possible. Returning to the example PIM application, all of the entities that make up the *Folder* and *User* screens are configuration information.

Group Entities into Business Objects Types

A Business Object Type represents a logical grouping of related configuration data. In the example application, it is clear that there are two concrete pieces of information.

They are as follows:

- Information pertaining to a *folder* and their related *permissions*
- Information specific to a *user* and its associated relative types such as *to-do* and *note* items.

Hence, the entities that make up the PIM application can be logically grouped into two categories – in other words, two Business Object Types. One is the *Folder* Business Object Type, which contains information specific to *folders* and their related *permissions*. The other is the *User* Business Object Type, which carries information about *users* and other related details.

Define Business Object Identifiers

Each Business Object must have a unique identifier. This is required by CTM in order to uniquely identify a Business Object within the system landscape.

The Business Object Identifier is comprised of the following elements:

- Business Object Type Identifier
- Business Object Key

Business Object Type Identifier

This is the identifier used by Business Object Modules to determine the type of the Business Object. Therefore, it must be unique for each Business Object Type and it must be possible for a Business Object Module implementation to identify the type of the Business Object from the identifier. It is important to ensure that this type name does not collide with any other Business Object Types available in the system, for example, it should not collide with the names of the Business Object Types provided out-of-the-box, or with any other Business Object Types developed by customer organizations. To guarantee this, it is recommended that Business Object Type names developed externally to the application are prefixed with a short abbreviation or name identifying the organization.

So, for an organization called "Sample Organization", the Business Object Type Identifier for the *Folder* Business Object Type could be *"so.Folder"*. Similarly, for the *User* Business Object Type; it could be *"so.User"*.

Business Object Key

This is the key used by Business Object Modules to uniquely identify an instance of a Business Object of a particular type. Where the Business Object supports versioning, the key should identify a particular version of a Business Object. It must be unique within a Business Object Type, and it must be possible for a Business Object Module to uniquely identify a Business Object in persistent storage using the key. To identify the Business Object Key for a Business Object, the implementation can choose to provide a combined value representing the primary key attributes of the Initial Entity. The Initial Entity is the root entity of a Business Object Type.

For further information, please refer to Analyzing the Folder Business Object Type on page 23.

The primary key attribute of the *Folder* Business Object Type's Initial Entity is *folderID*. So, the value of the *folderID* attribute can be used as the Business Object Key for *Folder* Business Objects. Similarly, the primary key attribute of the *User* Business Object Type's Initial Entity *User* is *userID*. So, the value of this attribute can be used as the Business Object Key for *User* Business Objects.

Note that it is also possible for an Initial Entity to have multiple primary key attributes. For example, take the example of a *Locale* Business Object Type with Initial Entity *Locale*. The *Locale* Entity includes the attributes *language*, *country*, and *variant*. To uniquely identify a particular *Locale*, the combined value of the attributes *language*, *country* and *variant* are required. So, the Business Object Key for the *Locale* Business Object must contain the values of these three attributes.

Ensure that the Configuration Entities use RAKS generated identifiers

As has been noted previously, it is mandatory for all configuration entities that form part of transportable Business Object Types to make use of the new Range Aware Key Server (RAKS) for the purposes of generating primary keys.

The Making Configuration Entities RAKS enabled on page 33 provides details of how the RAKS is used.

Analyzing the Folder Business Object Type

This section details the various steps involved in analyzing the configuration information for the *Folder* Business Object Type.

Identifying the Configuration Entities

The configuration entities that form the *Folder* Business Object Type are as follows:

- *Folder*
- *FolderPermission*
- *FolderPermissionLink*

Identifying the Initial Entity

The Initial Entity is the root entity of the Business Object Type. In other words, through this Initial Entity, it will be possible to identify all of the other entities in the logical grouping. For the *Folder* Business Object Type, the *Folder* entity is the Initial Entity. This is because from this entity, it is possible to identify the other entities in the Business Object, such as *FolderPermission* and *FolderPermissionLink*.

Identifying the Child Entities

All other entities in the logical grouping, excluding the Initial Entity are Child Entities. Hence, the entities *FolderPermission* and *FolderPermissionLink* become the Child Entities.

Identifying the Relative Entities

A Relative Entity refers to an entity whose information needs to be processed before processing a Child Entity. In this specific case, there are no Relative entities identified for the *Folder* Business Object Type.

Identifying dependencies

This section illustrates the varying level of dependencies that need to be identified as part of analysis for a Business Object Type.

The dependencies generally fall into two categories:

- Type-Level dependencies
- Instance-Level dependencies

Type-Level dependencies

Type-Level dependencies are applicable to relationships that arise after considering the possible set of values that a particular data field can contain. There is no need to examine the state of the data that comprise a Business Object in order to identify this level of dependency.

For example, consider the *type* attribute defined on the *Folder* entity. The value for this attribute comes from an entity called *CodeTableItem* which belongs to *CodeTable* Business Object Type. Hence, there exists a static dependency between the *Folder* and the *CodeTable* Business Object Types. The following diagram represents this type level dependency in UML.

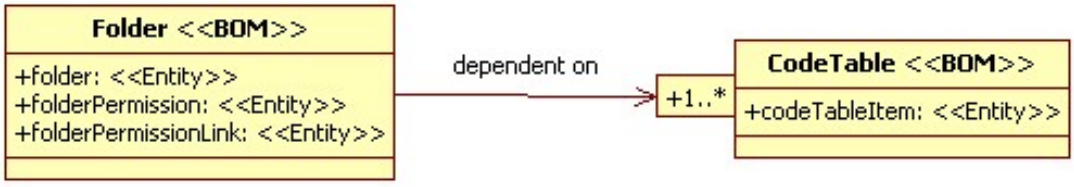

Figure 3: Type-level relationship between Folder BOM and CodeTable BOM

Another form of type-level dependency can be identified by examining the attributes of all entities in the Business Object Type. If any of the attributes have a foreign key relationship with the Initial Entity of another Business Object Type, then there is a type-level dependency on that Business Object Type. For example, assume that there is a Business Object Type called *File*. The Initial Entity of this Business Object Type is the *File* entity, which in turn has an attribute *folderID* with a foreign key relationship with the entity *Folder*. So, naturally, the *File* Business Object Type is related to the *Folder* Business Object Type. In other words, *File* has a type-level dependency on the *Folder* Business Object Type.Refer the following diagram that represents this relationship in UML

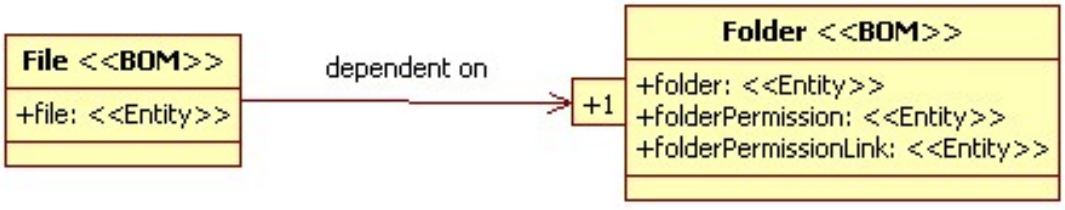

Figure 4: Type-level relationship between File BOM and Folder BOM

Instance-Level dependencies

Instance-Level dependencies are identified by examining the content of the Business Object.

This is best explained using examples:

Example 1

Consider a folder Business Object instance *'Documents/1'* with the following content:

```
Folder(1, 'Documents', 'Contains all documents', 'FT_PR, 'RST1')

FolderPermission(1, 'FPN_READ')

FolderPermissionLink(1,1,1)
```

After examining the data in each attribute, it is obvious that there are attributes that have instance relationships with *CodeTable* Business Objects. These are illustrated in the following table:

Attribute Name	Attribute Value	Dependent Business Object Instance
Folder.type	FT_PR	CodeTable/FolderType

Attribute Name	Attribute Value	Dependent Business Object Instance
Folder.statusCode	RST1	CodeTable/RecordStatus
FolderPermission.name	FPN_READ	CodeTable/ FolderPermissionName

The equivalent UML representation is shown in the following diagram.

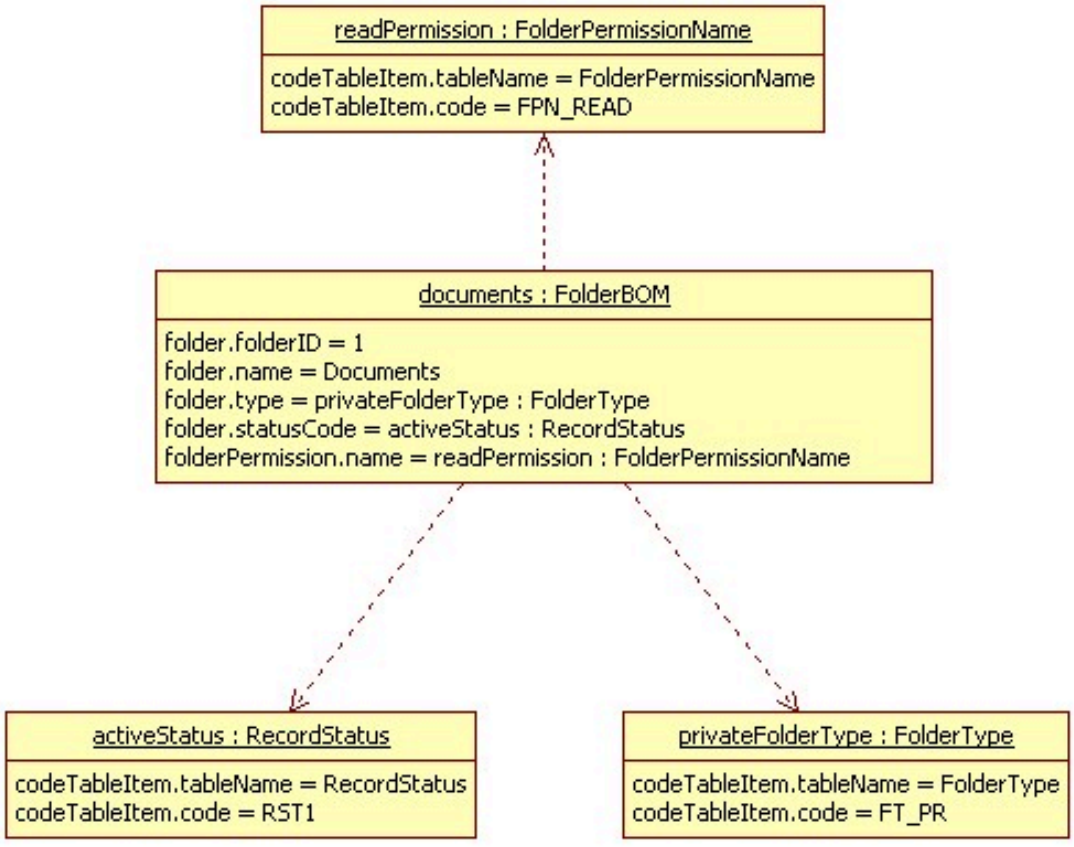

Figure 5: Instance-level relationships between Folder BOM and CodeTable BOM

Example 2

Consider a folder Business Object instance *'Pictures/2'* with the following contents:

```
Folder(2, 'Pictures', 'Contains all pictures', 'FT_PU, 'RST1')

FolderPermission(2, 'FPN_FC')

FolderPermissionLink(2,2,2)
```

After examining the data in each attribute, it is obvious that there are attributes that have instance relationships with *CodeTable* Business Objects. These are illustrated in the following table:

Attribute Name	Attribute Value	Dependent Business Object Instance
Folder.type	FT_PU	CodeTable/FolderType
Folder.statusCode	RST1	CodeTable/RecordStatus
FolderPermission.name	FPN_FC	CodeTable/ FolderPermissionName

The equivalent UML representation is shown in the following diagram.

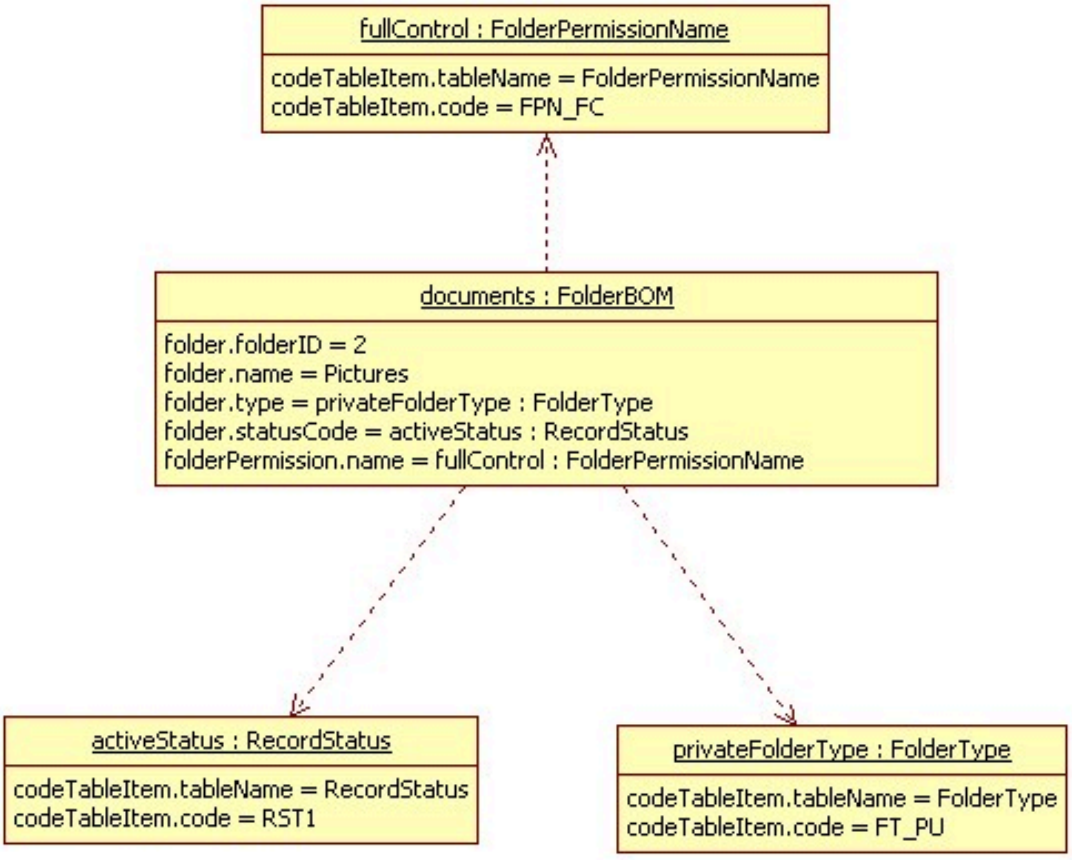

Figure 6: Instance-level relationships between Folder BOM and CodeTable BOM

Note that based on the content of the Business Object, the dependency information varies. Therefore, such dependencies are termed Instance-Level dependencies.

Identifying the Mode of Deletion

The Mode of Deletion refers to whether the Business Object Type is logically deleted or physically deleted. This is identified from the Mode of Deletion that is supported by the Initial Entity. In this case, the mode of deletion supported by the *Folder* entity is logical deletion; hence, the *Folder* Business Object Type supports logical deletion.

Analyzing the User Business Object Type

This section details the various steps that are involved in analyzing the configuration information for the *User* Business Object Type.

Identifying the configuration entities

The following configuration entities form the *User* Business Object Type:

- *User*
- *ToDo*
- *UserToDoLink*
- *Note*
- *UserNoteLink*

Identifying the Initial Entity

The entity *User* is the Initial Entity. This is because, starting with this entity, it is possible to identify both the *ToDo* and *Note* entities and the *UserToDoLink* and *UserNoteLink* link entities.

Identifying the Child Entities

All of the other entities in the logical grouping are child entities. Therefore, the entities *ToDo*, *Note*, *UserToDoLink* and *UserNoteLink* are all Child Entities.

Identifying the Relative Entities

A Relative Entity refers to an entity whose information needs to be processed before processing a child entity.

Identifying Relative Entities involves the following steps:

- Examine all attributes in all child entities for foreign key constraints.
- Determine the other entity upon which there is a constraint. This entity is known as the parent entity.
- Determine if the parent entity is a Relative Entity as follows:
    - If the parent entity is included in the same Business Object Type as the original entity, then identification of the Relative Entity can be ignored. This is because the parent entity will be processed in any case as part of the Business Object Type.
    - If the parent entity is not in the same Business Object Type as the original entity:
        - If the parent entity is the Initial Entity of another Business Object Type, then it is not a Relative Entity. Instead, the other Business Object Type has to be made a dependent Business Object Type.
        - If the parent entity is a Child Entity or a Relative Entity in another Business Object Type, then it should be considered a Relative Entity.

In the example application, working through the above steps, it is clear that there is only one Relative Entity for the *User* Business Object Type. This Relative Entity is *Category*. This is because the attribute *ToDo.categoryID* has a foreign key relation to the entity *Category*. However, let's imagine that there is a *Category* Business Object Type which has *Category* as the Initial

Entity; in this case, instead of *Category* being a Relative Entity, the *User* Business Object Type should declare the *Category* Business Object Type as a dependent Business Object Type.

Identifying dependencies

This section illustrates the dependencies that need to be identified when analyzing the *User* Business Object Type.

Type-Level dependencies

After studying the attributes of all of the entities and relationships, it is clear that the Business Object Type is dependent on the *CodeTable* and *Folder* Business Object Types.

The following table summarizes this information:

Attribute Name	Relationship Type	Dependent on
User.statusCode	CodeTable	RecordStatus
ToDo.statusCode	CodeTable	RecordStatus
ToDo.groupName	CodeTable	GroupName
Note.folderID	Foreign Key Constraint	Folder.folderID
ToDo.categoryID	Foreign Key Constraint	Category.categoryID

The following diagram shows the equivalent UML representation.

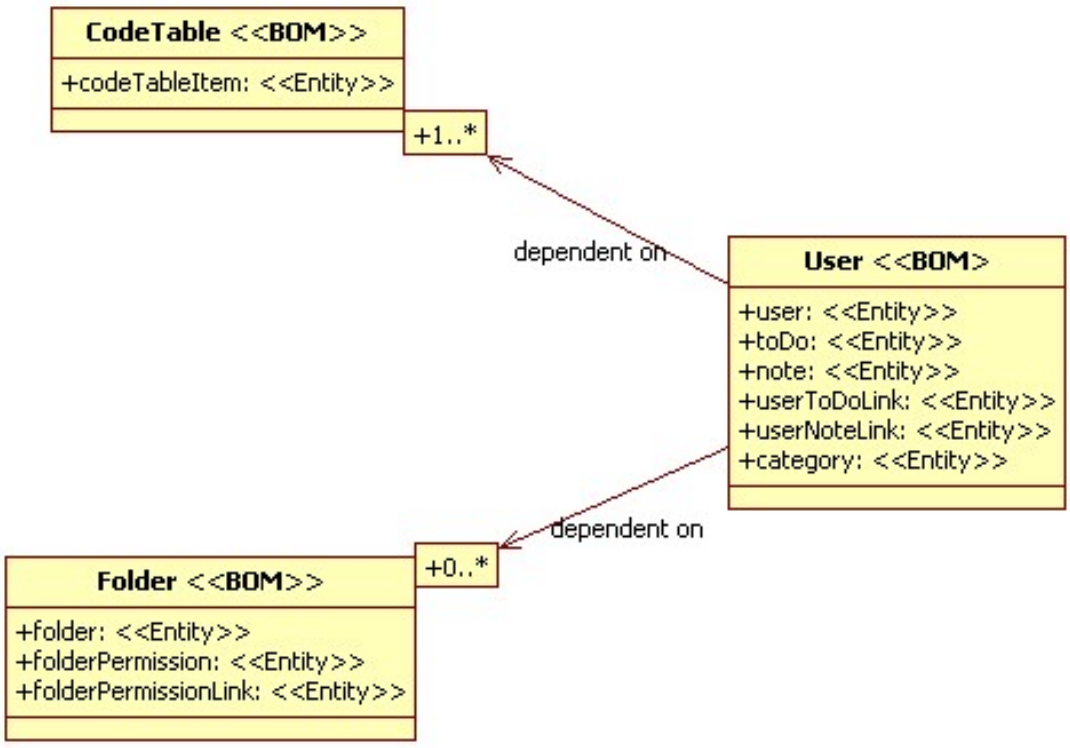

Figure 7: Type-level relationships for User BOM

Instance-Level dependencies

Instance-Level dependencies are identified by examining the content of the Business Object.

This is best explained using examples:

Example 1

Consider a user Business Object instance *'Admin/1'*, with the following contents:

```
User(1, 'Admin', 'Admin', 'RST1')

Category(1, 'CN_BIZ', 'Category denoting mail items')

ToDo(1, 'Approve mails', '2011-11-11', '2011-12-12', 1, '45',
 'RST1', 'GN_PRIV')

UserToDoLink(1, 1, 1)

Note(1, 'Send follow up mail', 'Send follow up mail', 1, 'RST1')

UserNoteLink(1, 1, 1)
```

After examining the data in each attribute, it is obvious that there are relationships with *CodeTable* and *Folder* Business Objects. These are listed in the following table:

Attribute Name	Attribute Value	Dependent Business Object Type/Identifier
User.statusCode	RST1	CodeTable/RecordStatus
Category.name	CN_BIZ	CodeTable/CategoryName
ToDo.statusCode	RST1	CodeTable/RecordStatus
ToDo.groupName	GN_PRIV	CodeTable/GroupName
Note.folderID	1	Folder/1

The equivalent UML representation is shown in the following diagram.

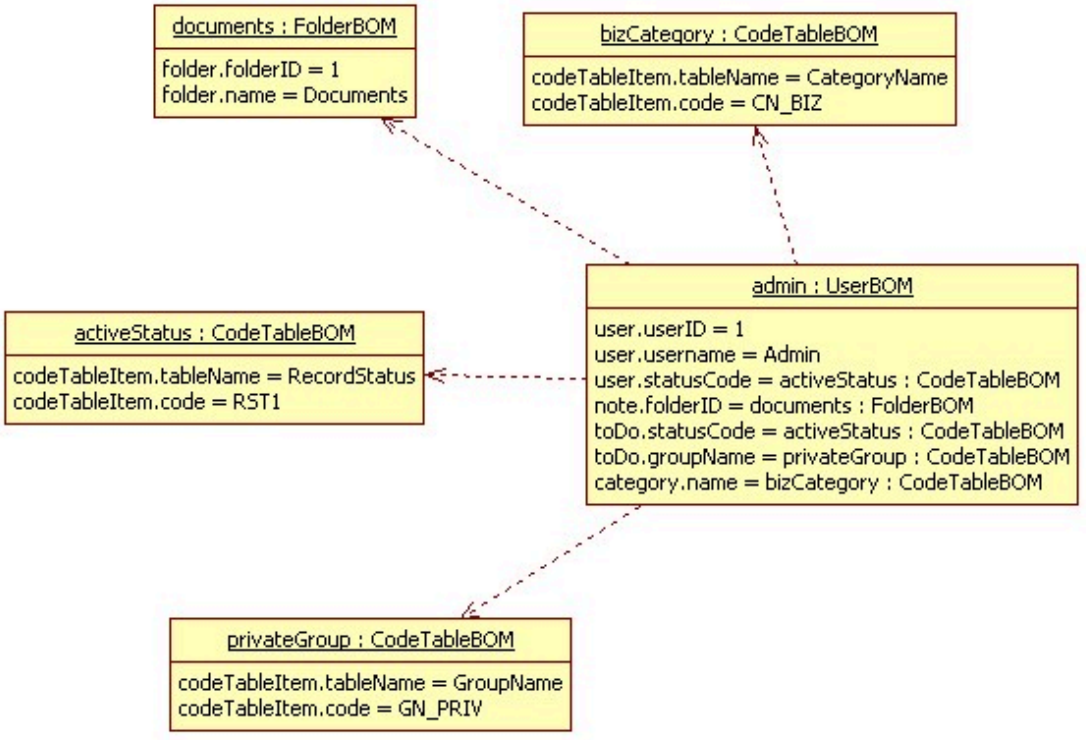

Figure 8: Instance-level relationships for User BOM

Example 2

Consider a folder Business Object instance, *'SuperAdmin/2'*, with the following contents.

```
User(2, 'SuperAdmin', 'SuperAdmin, 'RST1')
```

```
ToDo(2, 'Approve mails', '2011-11-11', '2011-12-12', null, '45',
 'RST1', 'GN_PRIV')

UserToDoLink(2, 2, 2)

Note(2, 'Send follow up mail', 'Send follow up mail', null,
 'RST1')

UserNoteLink(2, 2, 2)
```

In this case, there is a single instance level relationship, with a *CodeTable* Business Object. Observe that for the *Note* entity, the value of the attribute *folderID* is set to *NULL*. Therefore, in this case, the *'SuperAdmin/1'* Business Object is not related to the *Folder* Business Object.

Attribute Name	Attribute Value	Dependent Business Object Type/Identifier
User.statusCode	RST1	CodeTable / RecordStatus
Category.name	CN_BIZ	CodeTable / CategoryName
ToDo.statusCode	RST1	CodeTable / RecordStatus
ToDo.groupName	GN_PRIV	CodeTable / GroupName

The equivalent UML representation is shown in the following diagram.

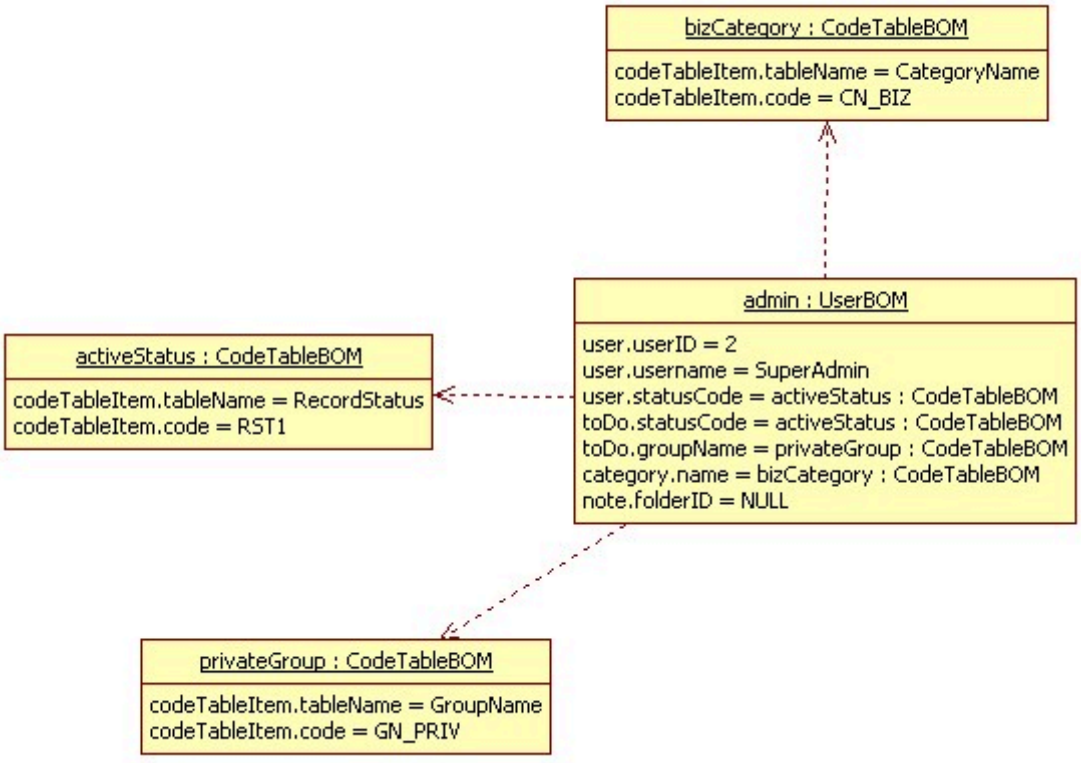

Figure 9: Instance-level relationships for User BOM

Identifying the Mode of Deletion

The Mode of Deletion refers to whether the Business Object Type is logically deleted or physically deleted. This is identified from the Mode of Deletion that is supported by the Initial Entity. In this case, the mode of deletion supported by the *User* entity is logical deletion; hence the *User* Business Object Type supports logical deletion.

Implementing BOMs

This section describes the steps involved in implementing the BOMs for a Business Object Type. The example *User* Business Object Type that was discussed in the previous sections is used to illustrated the process.

Broadly speaking, implementing BOMs involves the following:

- Making Configuration Entities RAKS enabled
- Writing the Entity Business Object Builders for all the entities
- Writing the BOM implementation

- Registering the BOM implementation(s) with the BOM registry

Making Configuration Entities RAKS enabled

Entities that form part of transportable Business Object Types must use primary keys generated by the Range Aware Key Server (RAKS).

The basic mechanism is to define a new Key Set for the Business Object Type, to configure the Key Set to supply keys generated by the RAKS, and then to move all entities in the Business Object Type to use keys from the Key Set. These keys will be generated using the RAKS.

Further details on the steps involved are provided in the following subsections. Again, the *User* Business Object Type that forms part of the example *pim* component is used to illustrate the process.

Creating New Key Set Configuration

Create a new file called *KeyServer.dmx* in the location *EJBServer\components\pim\data\initial* containing the key set definition for the Business Object Type.

Sample content for this file is provided below:

```
<row>
 <attribute name="strategy">
  <value>KB1002</value>
 </attribute>

 <attribute name="keySetCode">
  <value>UserBOMKS</value>
 </attribute>

 <attribute name="nextUniqueIdBlock">
  <value>0</value>
 </attribute>

 <attribute name="humanReadable">
  <value>1</value>
 </attribute>

 <attribute name="lastUpdated">
  <value>SYSTIME</value>
 </attribute>

 <attribute name="Annotation">
  <value>Key set for entities used in User BOM </value>
 </attribute>

</row>
```

There are several things to be noted in the Key Set definition:

- The property *strategy* with the value *KB1002* indicates to the Key Server that the RAKS implementation should be used to generate key values.
- The property *keySetCode* specifies the name of the Key Set. In order that the corresponding Business Object Type can be identified, it is recommended that this name is based on the Business Object Type name. So, in the example, the Key Set for the *User* BOM is named *UserBOMKS*
- The property *Annotation* is used to provide a description of the purpose of the Key Set configuration.

- Other properties such as *nextUniqueIdBlock*, *humanReadable* and *lastUpdated* are provided with sensible default values.

Ensure Entities use the new Key Set

All of the entities that form part of the Business Object Type must use keys generated using the new Key Set. This includes the Initial Entity, the Child Entities and the Relative Entities. So, in the example, all of the entities that constitute the *User* Business object must use keys generated using the *UserBOMKS* key set. This mechanism for achieving this depends on whether the entity uses Auto ID Generation, or invokes the Key Server directly.

Auto ID Generation

Most entities use the key server implicitly – that is, keys are generated automatically when insert operations are invoked. This is known as Auto ID generation. In order to configure these entities to use the RAKS, use the following procedure:

- Within the Rational® Software Architect modeling environment, identify all of the *insert* operations on the relevant entities. That is, identify the operations with the *insert* stereotype.
- For each insert operation, perform the following steps:
  - Go to Properties; navigate to the Curam tab. Populate the property *Auto ID Field* with the name of the primary key attribute. For example, for the sample entity the value for the *Auto ID Field* will be *userID*.
  - Following this, populate the property *Auto ID Key* with the name of the Key Set for the Business Object Type. So, in the sample application, the field value will be *UserBOMKS*.

Code that invokes the Key Server directly

The Key Server can also be invoked directly, in code. This is achieved by directly invoking on the one of the *curam.util.type.UniqueID* class' static methods – usually the *nextUniqueID()* method. Typically, code that invokes the standard Key Server will be along the following lines:

```
ToDoDtls toDoDtls = new ToDoDtls();
toDoDtls.toDoID = UniqueID.nextUniqueID();
```

In order to use the RAKS, this code should be changed as follows:

```
UniqueIDKeySet uniqueIDKeySet = new UniqueIDKeySet();
uniqueIDKeySet.keySetName = "UserBOMKS";

ToDoDtls toDoDtls = new ToDoDtls();
toDoDtls.toDoId = UniqueID.getNextIDFromKeySet(uniqueIDKeySet);
```

Business Object Classes

The following sub-sections of this document include several code snippets. These snippets assume that the certain classes are available for the Business Object Type – for example. façade classes, DAO classes, and so on.

For full details of these classes, please refer 1.4 Assumptions on the availability of classes on page 45

Developing Entity Business Object Builder classes for the entities

This section describes the steps involved in writing the Entity Business Object Builder classes for all of the entities in the example *User* Business Object Type. Please refer to BOM Infrastructure on page 13 for information on the purpose of Entity Business Object Builder classes.

Category Entity Business Object Builder

The implementation of Entity Business Object Builder for the *Category* entity is reasonable straightforward.

Class declaration

The class must extend from *curam.ctm.bom.util.impl.AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *CategoryDtls* respectively. Therefore, these types must be specified in the class declaration as follows:

```
class CategoryEntityBOBuilder
 extends AbstractEntityBOBuilder<Long, CategoryDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden to return the name of the entity that is unique across the system - i.e. in this case, the string *Category* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method *getEntityAdapter()* will be called by the infrastructure in order to obtain the entity's Adapter class to perform various operations on the entity. Hence this method has to be overridden to return an instance of *CategoryAdapter* class.

Note Entity Business Object Builder

Class declaration

The class must extend from *AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *NoteDtls* respectively. Hence, these types must be specified in the class declaration as follows,

```
class NoteEntityBOBuilder extends
 AbstractEntityBOBuilder<Long, NoteDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden to return the name of the entity that is unique across the system - i.e. in this case, the string *Note* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method *getEntityAdapter()* has to be overridden in order to return an instance of the adapter class - i.e. *NoteAdapter*.

User Note Link Entity Business Object Builder

Class declaration

The class must extend from *AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *UserNoteLinkDtls* respectively. Hence these types must be specified in the class declaration as follows:

```
class UserNoteLinkEntityBOBuilder
```

```
 extends AbstractEntityBOBuilder<Long, UserNoteLinkDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden to return the name of the entity that is unique across the system - i.e. in this case, the string *UserNoteLink* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method *getEntityAdapter()* has to be overridden to return an instance of the adapter class - i.e. *UserNoteLinkAdapter*.

ToDo Entity Business Object Builder

Class declaration

The class must extend from *AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *ToDoDtls* respectively. Hence these types must be specified in the class declaration as follows,

```
class ToDoEntityBOBuilder extends
 AbstractEntityBOBuilder<Long, ToDoDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden to return the name of the entity that is unique across the system - i.e. in this case, the string *ToDo* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method getEntityAdapter() has to be overridden to return an instance of the adapter class - i.e. ToDoAdapter.

User ToDo Link Entity Business Object Builder

Class declaration

The class must extend *AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *UserToDoLinkDtls* respectively. Hence, these types must be specified in the class declaration as follows:

```
class UserToDoLinkEntityBOBuilder extends
 AbstractEntityBOBuilder<Long, UserToDoLinkDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden in order to return the name of the entity that is unique across the system - i.e. in this case, the string *UserToDoLink* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method *getEntityAdapter()* has to be overridden to return an instance of the adapter class, i.e. *UserToDoLinkAdapter*.

User Entity Business Object Builder

Details of the implementation of Entity Business Object Builder for the *User* entity are provided below:

Class declaration

The class must extend from *AbstractEntityBOBuilder*. The primary key data type and the entity's *Dtls* data type are *Long* and *UserDtls* respectively. Hence these types must be specified in the class declaration as follows,

```
class UserEntityBOBuilder extends
 AbstractEntityBOBuilder<Long, UserDtls>{
}
```

getName() : Returning the name of the entity

The method *getName()* has to be overridden to return the name of the entity that is unique across the system - i.e. in this case, the string *User* can be returned.

getEntityAdapter() : Returning the Entity Adapter instance

The method *getEntityAdapter()* has to be overridden to return an instance of the adapter class, i.e. *UserAdapter*.

Implementing the BOMs

This section describes the steps involved in implementing the various BOMs for a Business Object Type. The most straightforward mechanism is to develop a single BOM class for all of the BOM interfaces that are required for the Business Object Type. That is, the class implements all of the required BOM interfaces. The recommended means of doing this is to extend the BOM Infrastructure class *curam.ctm.bom.util.impl.AbstractBOM*, which implements the BOM interfaces, and provides out-of-the-box implementations of several of the methods.

Note, however, that it is also possible to implement all of the BOM interfaces directly if desired.

The process of implementing the BOM is now described through example, using the sample *User* Business Object Type described above. In the example, the recommended strategy of providing a single BOM implementation class is followed.

Extend AbstractBOM

The first step is to extend the *curam.ctm.bom.util.impl.AbstractBOM* class:

```
public class UserBOM extends AbstractBOM {

//  Provide BOM implementation methods

}
```

Singleton BOMs

An important factor to bear in mind when developing the class is that BOM implementations are singletons. That is, a single BOM instance will be created and used for all processing of a Business Object Type. So, if two Change Sets containing instances if the same Business Object Type are being processed at the same time, the same BOM instance will be used. BOM implementations must therefore be capable of being used by multiple threads simultaneously, that

is, must be thread-safe. The best mechanism for achieving this is to avoid storing Business Object (instance) -specific state in BOM implementations.

AbstractBOM Method Implementations

The next step is to implement the BOM methods. Details on how to do this are provided. Note also that the Javadoc for the BOM interfaces provides more information on each of the methods. These interfaces are all members of the package *curam.util.ctm.bom*.

getName() : Retrieving the name of the Business Object Type

The method *getName()* should return the name of the Business Object Type. This name will be displayed in the User Interface while searching for the set of Business Object Types available on a system. For example, the implementation for *User* BOMs could return the BOM name *User*.

getInitialBO() : Provide the Entity Business Object Builder for the Initial Entity

The get *InitialBO()* method implementation should provide the Entity Business Object Builder for the Initial Entity of the Business Object Type. For the sample *User* Business Object Type, this is the *User* entity. The following code snippet illustrates the process:

```
protected AbstractBOBuilder getInitialBO(
 final BusinessObjectIdentifier boIdentifier) {

 final UserEntityBOBuilder userEntityBOBuilder
  = userEntityBOBuilderProvider.get();

 userEntityBOBuilder.setID(Long.parseLong(
  boIdentifier.getBusinessObjectKey().get()));

 return userEntityBOBuilder;

}
```

In the code snippet, a new instance of *UserEntityBOBuilder* is created and initiated with the identifier obtained from the incoming *BusinessObjectIdentifier*. This instance is then returned.

The BOTraits annotation: Specifying the Mode of Deletion

The class level annotation *curam.util.ctm.bom.annotation.BOTraits* is used to indicate the Mode of Deletion supported by this Business Object Type. It needs to be specified on the implementation of the *curam.util.ctm.bom.InformationalBOM* interface. In the *UserBOM* example, a common implementation class is being developed for all BOMs (i.e. the *UserBOM* class). So the annotation is specified on this class. This is illustrated on the following code snippet:

```
@BOTraits(deletionMode = DeletionMode.LOGICAL)
public class UserBOM extends AbstractBOM{

}
```

The annotation in the example declares that the *User* Business Object Type supports logical deletion. However, note that if the *BOTraits* annotation is not specified, the infrastructure assumes that the Business Object Type uses Logical deletion. Hence for a Business Object Type that uses Logical deletion, it is not mandatory to provide this annotation. However, Business Object Types that are physically deleted must specify the annotation, using the deletion mode *DeletionMode.PHYSICAL*.

getDependentBusinessObjectIdentifiers() : Fetching the Dependent Business Object identifiers

The *getDependentBusinessObjectIdentifiers()* method implementation should return the set of Business Object identifiers on which the Business Object is dependent (if any). The following code snippet illustrates the process:

```
public Set<BusinessObjectIdentifier>
 getDependentBusinessObjectIdentifiers(
 final BusinessObjectIdentifier boIdentifier){

 final Set<BusinessObjectIdentifier> setOfDependantBOs
     = new HashSet<BusinessObjectIdentifier>();

 // Adding CodeTable dependencies.
 addCodeTableBusinessTypeDependency(
     setOfDependantBOs, RECORDSTATUSEntry.TABLENAME);
 addCodeTableBusinessTypeDependency(
     setOfDependantBOs, GroupNameEntry.TABLENAME);
 addCodeTableBusinessTypeDependency(
     setOfDependantBOs, CategoryName.TABLENAME);

     // Add Folder dependencies
 final User user = userDAO.get(Long.parseLong(
     boIdentifier.getBusinessObjectKey().get()));

 for (final Note note : userDAO.searchAllNotes(user)){

  final Folder folder = note.getFolder();

            setOfDependantBOs.add(
                  BusinessObjectIdentifierFactoryImpl.get().
                        createBusinessObjectIdentifier(
                  FolderBOMConstants.kFolderBusinessObjectType.
                  get(), String.valueOf(folder.getID())));

 }
 return setOfDependantBOs;

}
```

As previously noted, during the Business Object Type analysis, it was identified that the *User* Business Object is dependent on the *CodeTable* and *Folder* Business Objects. Therefore, the code snippet above adds the relevant *CodeTable* Business Objects as dependencies. This is achieved by calling the method *addCodeTableBusinessTypeDependency()*. Additionally, because the *User* entity can be related to the *Folder* entity through *Note* entity, the code calls *searchAllNotes()* to retrieve the set of *Note* entities related to a *user*. Then, for each *Note*, the corresponding *Folder* Business Object is identified and added to the set to be returned.

getReadSecurityIdentifier() : Retrieving the Read Security identifiers

The *getReadSecurityIdentifier()* method implementation has to return all of the security identifiers (SIDs) required to read the Business Object content. This is used to assess whether or not an administrative user using CTM has the required read permissions for the Business Object. An example code snippet is provided below:

```
public public Set<String> getReadSecurityIdentifier() {
```

```
    final Set<String> readSecurityIdentifiers
     = new HashSet<String>();

    readSecurityIdentifiers.add("UserManager.readUser");
    readSecurityIdentifiers.add("UserManager.readAllNotes");
    readSecurityIdentifiers.add("UserManager.readAllToDos");
    readSecurityIdentifiers.add("UserManager.readAllToDos");
    readSecurityIdentifiers.add("NoteManager.readNote");
    readSecurityIdentifiers.add("ToDoManager.readToDo");

    return readSecurityIdentifiers;

  }
```

The above implementation gathers together all of the read operation SIDs from the relevant Façade APIs. Refer to 1.4 Assumptions on the availability of classes on page 45 for more details.

getWriteSecurityIdentifier() : Retrieving the Write Security identifiers

Similarly, the method *getWriteSecurityIdentifier()* needs to specify all of the security identifies (SIDs) required to write the Business Object content. An example code snippet is provided below:

```
public Set<String> getWriteSecurityIdentifier() {

 final Set<String> writeSecurityIdentifiers
  = new HashSet<String>();

 writeSecurityIdentifiers.add("UserManager.createUser");
 writeSecurityIdentifiers.add("UserManager.editUser");
 writeSecurityIdentifiers.add("UserManager.deleteUser");
 writeSecurityIdentifiers.add("UserManager.associateNotes");
 writeSecurityIdentifiers.add("UserManager.disassociateNotes");
 writeSecurityIdentifiers.add("UserManager.associateToDos");
 writeSecurityIdentifiers.add("UserManager.disassociateToDos");

 writeSecurityIdentifiers.add("NoteManager.createNote");
 writeSecurityIdentifiers.add("NoteManager.editNote");
 writeSecurityIdentifiers.add("NoteManager.deleteNote");

 writeSecurityIdentifiers.add("ToDoManager.createToDo");
 writeSecurityIdentifiers.add("ToDoManager.editToDo");
 writeSecurityIdentifiers.add("ToDoManager.deleteToDo");

 return writeSecurityIdentifiers;
}
```

The above implementation gathers together all of the write operation SIDs from the relevant Façade APIs. Refer to 1.4 Assumptions on the availability of classes on page 45 for more details.

Registering the BOM implementation

The next step is to register the BOM implementation(s) with the BOM registry. The BOM registry, implemented using Guice, acts as a central access point for the CTM Infrastructure to

obtain BOM implementations. So, the *User* BOM implementation has to be registered with the BOM registry. The following sections detail the steps involved in registering BOMs.

Again, the process is illustrated by example, using the sample *UserBOM*:

Create a new Guice Module class

Create a new Guice Module called *UserBOMModule* that extends from *com.google.inject.AbstractModule* and provide an implementation of the configure() method as follows,

```
protected void configure() {

 final Multibinder<ReadAndUpsertBOM> readAndUpsertBOMBinder
  = Multibinder.newSetBinder(binder(),
    ReadAndUpsertBOM.class);
 readAndUpsertBOMBinder.addBinding().to(UserBOM.class);

 final Multibinder<InformationalBOM> informationalBOMBinder
  = Multibinder.newSetBinder(binder(),
    InformationalBOM.class);
 informationalBOMBinder.addBinding().to(UserBOM.class);

 final Multibinder<SecurityBOM> securityBOMBinder
  = Multibinder.newSetBinder(binder(),
    SecurityBOM.class);
 securityBOMBinder.addBinding().to(UserBOM.class);

 final Multibinder<DeleteBOM> deleteBOMBinder
  = Multibinder.newSetBinder(binder(),
    DeleteBOM.class);
 deleteBOMBinder.addBinding().to(UserBOM.class);

 final Multibinder<DependentBOM> dependentBOMBinder
  = Multibinder.newSetBinder(binder(),
    DependentBOM.class);
 dependentBOMBinder.addBinding().to(UserBOM.class);

 final Multibinder<ExistenceBOM> existenceBOMBinder
  = Multibinder.newSetBinder(binder(),
    ExistenceBOM.class);
 existenceBOMBinder.addBinding().to(UserBOM.class);

}
```

Note that in the above code snippet, a new *com.google.inject.multibindings.Multibinder* instance is created in order to hold multiple implementations of the *curam.util.ctm.bom.ReadAndUpsertBOM* interface. An object of type *UserBOM* is bound to this binder using the standard *addBinding()* method. The process is repeated with binders for all of the other BOM types - i.e. for *curam.util.ctm.bom.InformationalBOM*, *curam.util.ctm.bom.SecurityBOM*, *curam.util.ctm.bom.DeleteBOM*, *curam.util.ctm.bom.DependentBOM* and *curam.util.ctm.bom.ExistenceBOM* interfaces. Note that as a single implementation is used for all of the BOM types, the same class is bound to each of the binders (i.e. *UserBOM*).

Update the new Module class in the MODULECLASSNAME DMX file

Each component can have a *MODULECLASSNAME.dmx* DMX file containing the configuration information for the component's *Module* classes (if any). The fully-qualified class name of the Module registering the BOMs must be placed in this file.

For the *pim* component, the file path of the DMX file will be *EJBServer\components\pim\data\initial\MODULECLASSNAME.dmx*. This file will need to contain the following information:

```
<row>
 <attribute name="moduleClassName">
  <value>sample.package.UserBOMModule</value>
 </attribute>
</row>
```

In the code snippet above, the value of the child element *value* must be the fully qualified name of the Module class - i.e. *sample.package.USERBOMModule* in this case.

Optional BOM types

This section covers the optional BOM types and briefly explains their purpose. Any optional BOMs required for a Business Object Type should be implemented, adhering to the appropriate contract described in the BOM Javadoc.

The implementations should then be registered with the BOM registry, using the same pattern documented above, that is, in the Registering the BOM implementation on page 40.

Pre Commit Action BOM

This BOM is used to perform any pre-processing actions on a Business Object during an *Apply* operation before the Change Set is committed to the database. An example of an activity that could be implemented in this BOM is validating the Business Object contents against other Business Objects that may have been in the Change Set. This BOM can be implemented by providing an implementation of the interface *curam.util.ctm.bom.PreCommitActionBOM*.

Please refer to the Javadoc for *curam.util.ctm.bom.PreCommitActionBOM* for further information.

Pre Commit Action Type BOM

This BOM is used for pre-processing actions required at a Business Object Type – level during an *Apply* operation before the Change Set is committed to the database. This means that irrespective of the number of the Business Object instances available for a particular Business Object Type in a Change Set, the BOM implementation will be called only once. This BOM can be implemented by providing an implementation of the interface *curam.util.ctm.bom.PreCommitActionTypeBOM*.

Please refer to the Javadoc for *curam.util.ctm.bom.PreCommitActionTypeBOM* for further information.

Post Commit Action BOM

A BOM for performing any post processing actions after the transaction for an *Apply* operation has been committed. This BOM can be implemented by providing an implementation of the interface *curam.util.ctm.bom.PostCommitActionBOM*. Note that unlike the other BOMs, a separate transaction is used for *curam.util.ctm.bom.PostCommitActionBOM* implementations, and that the BOMs are invoked after the Apply transaction has been committed. Therefore, again, unlike the other BOMs, implementations of this BOM cannot terminate the Apply process by rolling back the transaction.

Please refer to the Javadoc for *curam.util.ctm.bom.PostCommitActionBOM* for further information on this BOM.

Revert Change Set Construction Handler BOM

Business Object Types that need to customize the process of constructing a *revert Change Set* can achieve this by providing an implementation of this BOM. This BOM can be implemented by providing an implementation of the interface *curam.util.ctm.bom.RevertChangeSetConstructionHandlerBOM*.

Please refer to both the 1.5 Customizing the construction of revert Change Set on page 47 in the Appendix and to the Javadoc for *curam.util.ctm.bom.RevertChangeSetConstructionHandlerBOM* for more details.

Authorisation BOM

In order to verify whether or not an administrative user can access a Business Object, it is generally sufficient to check that a user has the SIDs that are required to read and write instances of the Business Object Type. As noted above, the SIDs required for a particular Business Object are provided to CTM by implementing the *curam.util.ctm.bom.SecurityBOM*.

However, some Business Object Types may have more advanced security requirements, involving custom programmatic security checks. These checks can be implemented in *curam.util.ctm.bom.AuthorisationBOM* for the Business Object Type. If *curam.util.ctm.bom.AuthorisationBOM* is provided for a Business Object Type, it will be used instead of the *curam.util.ctm.bom.SecurityBOM* to verify whether or not a user can read or write instances of the Business Object Type.

This BOM can be implemented by providing an implementation of the interface *curam.util.ctm.bom.AuthorisationBOM*. Please refer to the Javadoc for *curam.util.ctm.bom.AuthorisationBOM* for further information.

Testing the transport of Business Object Types

This section discusses the common testing scenarios that are applicable to most Business Object Types.

Pre-requisites

The following should be setup before testing commences:

- The source and the target systems should both be available, and should both be configured to be in the same system landscape.
- The target system should be configured as a destination system for transport purposes within the source system.
- Configuration data should be available for all of the entities that form the Business Object Type being tested.

Testing the User Business Object Type via the Administrative User Interface

It is important to carefully test the BOM implementations. In order to do this, a comprehensive set of unit tests should be developed for the BOM implementations, and the functionality should be thoroughly tested via the user interface.

The process of testing Business Object Types via the Administrative User Interface is described in the following sub-sections. Again, the procedure is illustrated using the example *User* Business Object Type.

Listing all active User objects

Create a new Change Set, locate the *User* Business Object Type and search for the available Business Objects. The screen should only list the *User* objects that are active.

Checking the dependent Business Objects

Populate a new Change Set with sample *User* Business Objects. Expand the Change Set and select any of the *User* Business Objects. Select the option *Add Related Business Object*. A pop-up window showing the Related Business Objects will open and it should contain *CodeTable/RecordStatus*, *CodeTable/GroupName* and *CodeTable/CategoryName* items. If the selected Business Object has any dependency on a *Folder* Business Object, then the Folder Business Object instance must also be listed.

Releasing the Change Set

Select the *Release* option for the Change Set. Ensure that the *Release* operation completes successfully, that is, without any errors.

Exporting the Change Set

Select the *Export* option on the released Change Set. This export option will convert the contents of all of the Business Objects in the Change Set into XML format. The *Export* operation must successfully complete without any errors.

Transporting the Change Set

Select the *Transport* option of the release Change Set. Specify the target machine as the destination to which the Change Set should be transported. The *Transport* operation must successfully complete without any errors. Navigate to the target system's CTM screens to verify that the transported Change Set is available.

Apply the Change Set

Select the *Apply* option on the released Change Set on the target system. This operation will commit the content of the Business Objects from the Change Set to the target system. To check the availability of the Business Objects, navigate to the home page of *Folder* and *User* to check if the Business Objects transported from the source machine are listed.

Undoing the Change Set

Select the newly applied Change Set on the target system. Choose the *Undo* operation. This operation will "undo" the Change Set, that is, the content of the Business Objects that were previously applied will instead be reverted to their old values, or deleted if they were not already present on the target system. The delete will either be a logical delete or a physical delete, as appropriate to the Business Object Type. To verify the correctness of the *Undo* operation, navigate to the home page of *Folder* and *User* to check if the Business Objects are in the *Inactive* state, that is, have been logically deleted.

1.4 Assumptions on the availability of classes

The examples in code assume that the following classes are available for the Business Object type:

Availability of Facade APIs for managing user operations

It is assumed that there are Facade APIs which provide CRUD operations for the *user* entity, and provide functionalities for associating/disassociating *to-do* and *note* entities from *users*. Equivalent Facade APIs are also available for *to-do* and *note* objects.

The table below provides details of the operations:

Facade Name	Operation Name	Description
UserManager	createUser	Creates a new user
	editUser	Modify details on existing user
	readUser	Reads and returns user information
	deleteUser	Removes the user from the system
	associateNotes	Associates note items to a user
	disassociateNotes	Dis-associates note items from a user
	readAllNotes	Fetches all the notes for a user
	associateToDos	Associates to-do items with a user
	disassociateToDos	Disassociates to-do items from a user
	readAllToDos	Fetches all the to-do items for a user
NoteManager	createNote	Creates a new note
	editNote	Edits information from existing note
	deleteNote	Removes the note
	readNote	Reads and returns note information
ToDoManager	createToDo	Creates a new to-do
	editToDo	Edits information from existing to-do
	deleteToDo	Removes the to-do

Facade Name	Operation Name	Description
	readToDo	Reads and returns to-do information

Availability of Adapter classes

It is assumed that the PI Adapter classes are generated and available for the Initial Entity, Child Entities and Relative Entities.

This means that for the *User* BOM, the adapter classes described below are available:

- *UserAdapter*
- *NoteAdapter*
- *UserNoteLinkAdapter*
- *ToDoAdapter*
- *UserToDoLinkAdapter*
- *CategoryAdapter*

Availability of Data Access Object classes

In most cases, Adapter classes are sufficient to perform database related operations. However, it is possible that there are entities related to other entities through foreign key associations and, in such cases, it is desirable to provide Data Access Classes to facilitate fetching data from multiple entities.

For instance, for the *User* Business Object Type, at least one Data Access Object class is required:

- *UserDAO*, to fetch related information for a *user* from *to-do* and *note* items

DAO Class name	Operation Name	Description
UserDAO	searchAllUsers	Searches all the users in the system
	searchAllToDos	Search all the related to-do items for a user
	searchAllToDoLinks	Search all the related to-do item links for a user
	searchAllNotes	Search all the related note items for a user
	searchAllNoteLinks	Search all the related note item links for a user
	searchAllCategories	Search all the related categories for notes that are associated with a user

Availability of classes generated from Code Tables

The entities that form the *User* Business Object Type are dependent on several code tables. Hence, it is assumed that the equivalent Java® classes for these code tables are also available.

Refer to the table below for details of these classes:

Code Table Name	Java Identifier
GroupName	GroupNameEntry
CategoryName	CategoryNameEntry

1.5 Customizing the construction of revert Change Set

You might want to customize the content of a revert Change Set. To illustrate a scenario when would be necessary, assume that there can be only one active *User* entity in the system. This means that when a new *User* is applied to the system, the previously active *User* entity becomes inactive, and the current one becomes active.

As an example, consider a Change Set containing a new Business Object *User/X*. Also, assume that in the database, there is already an active *User* Business Object instance, *User/A*. While applying the Change Set, the infrastructure creates a corresponding revert Change Set. This revert *Change Set* contains a *Delete* instruction for the newly added User. When the original Change Set is applied, *User/X* will be active, and *User/A* is inactive.

Now, when applied, the revert Change Set should ideally de-activate *User/X* (as this was newly created through the Change Set) and reactivate *User/A* (as this was previously active). However, this is only possible if the revert Change Set contains the following instructions:

- *User/X-Delete*
- *User/A-Add*

While the *revert Change Set* automatically contains the instruction *User/X-Delete* (as it is newly created by applying the original Change Set), it does not contain the instruction *User/A-Add*. This is because this Business Object was not in the original Change Set, and is not directly related to the *User/X* Business Object in the original Change Set. So, the implementer of the BOMs for *User* has to identify the unrelated Business Object(s) (that is, *User-A* in this case), and ensure that it is placed in the *revert Change Set*.

To provide this functionality, an implementation of *curam.util.ctm.bom.RevertChangeSetConstructionHandlerBOM* must be developed. This BOM requires an implementation of the *constructBusinessObjectIdentifiers()* method that returns the identifiers of unrelated Business Object that are required for revert purposes.

Note that there is no need for the implementation to specify the instruction type of the unrelated Business Object(s), because the infrastructure knows how to identify the instruction type for a specific Business Object identifier

The code snippet here illustrates the implementation of the *curam.util.ctm.bom.RevertChangeSetConstructionHandlerBOM* 's *constructBusinessObjectIdentifiers()* for the *User* Business Object Type:

```
public final Set<BusinessObjectIdentifier>
```

```
constructBusinessObjectIdentifiers(
 BusinessObjectIdentifier boIdentifier,
  Document boDocument) {

Set<BusinessObjectIdentifier> boIdentifiers
     = new HashSet<BusinessObjectIdentifier>();
BusinessObjectIdentifier activeBO
 = getActiveBusinessObjectIdentifier();

if (activeBO!= null) {

 // If the identified active Business Object is equal
 // to the incoming Business Object identifier, then
 // we should not have to include it, because this
 // Business Object identifier would have
 // already undergone processing by the framework.

 if (!activeBO.equals(boIdentifier)) {

  // Active Business Object exists in the
  // database. This needs to be included
  // in the revert Change Set XML.

  boIdentifiers.add(activeBO);
 }
}
return boIdentifiers;

}
```

1.6 Reference guides

-
-
-
- Google Guice 2

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this Merative product and use of those websites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Merative, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.