IBM Cúram Social Program Management

Cúram Modeling Reference Guide

Version 6.0.4

Note

Before using this information and the product it supports, read the information in Notices at the back of this guide.

This edition applies to version 6.0.4 of IBM Cúram Social Program Management and all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Chapter 1

Introduction

1.1 Overview

IBM Cúram Social Program Management enables the creation of client-server applications by minimizing the complexity application developers face in developing database access, EJB management, client-server interaction, etc. This minimizing of complexity is achieved by developers modeling the application(s) they wish to create using a UML[1] meta-model.

The *Cúram Generator* uses this UML meta-model to automatically generate all the required stubs, skeletons, classes, and communications required to interact with a back-end database and remote clients leaving the developer to concentrate on providing the application business logic. This guide describes the tools and components available to an application developer when they are modeling the application, and how the *Cúram Server Code Generator* will treat each of these components when generating classes.

The UML meta-model is a platform-independent model which describes the following aspects of the application:

- *Domains*—application-specific datatypes. Analogous to C++ typedefs.
- *Entities*—the objects modelled and persistently stored by the application. These correspond to relational database tables.
- *Processes*—related sets of activities to achieve some business goal.
- *Structs*—passed as messages throughout the application. Analogous to structs in C++.
- *Remote Interfaces*—client-visible interfaces through which server functionality may be accessed.

This document provides a reference for Cúram model-based functionality such as: Cúram domains, classes, operations, attributes and how they map to the underlying database.

The model is edited using the *IBM® Rational® Software Architect* and code is generated using the Cúram Server Code Generator.

1.2 Intended Audience

This document should be read by people who will be using the Cúram Server Development Environment for *Java®* (SDEJ) tools to generate applications from UML models.

1.3 Prerequisites

The reader should be familiar with Cúram model development concepts and should have a good working knowledge of the following before reading this document:

- Cúram server development (see the *Cúram Server Developer's Guide* for more details);
- *Rational Software Architect*;
- SQL;
- UML;
- *Java*.

1.4 Rational Software Architect

Rational Software Architect is the third-party tool used for developing and maintaining the UML meta-model. It is primarily used as a tool for object mapping, analysis and design. A key reason *Rational Software Architect* was selected for these functions is because of its extensibility, which enables support of modeling for Cúram. The use of *Rational Software Architect* is covered in the *Working with the Cúram Model in Rational Software Architect* document.

Note

Please refer to the *Cúram Supported Prerequisites* document for more information on the supported versions of third party tools.

1.5 The Cúram Server Code Generator

The *Cúram Server Code Generator* takes as its input the UML meta-model and produces the following outputs:

- *Java* server implementation code;
- *Java* beans;

- XML[2] for the database entities and other classes in the model.

1.6 Chapters in this Guide

This chapters in this guide can be logically grouped into six parts as described below.

1.6.1 Part 1 - UML Overview

Part 1 consists of a high-level overview of UML and how it applies to Cúram modeling. These are the chapters making up this part:

- Chapter 2 UML Overview
- Chapter 3 Packages

1.6.2 Part 2 - Class Stereotypes

Part 2 provides reference and usage information for the various class stereotypes used with Cúram modeling and related topics.

Some notes applicable to classes in general:

- Two or more struct or process classes may have the same name provided they are in different packages and have different `CODE_PACKAGE` names. All other class names must be completely unique within the input model. For more information on the `CODE_PACKAGE` option see the CODE_PACKAGE option in Section 3.2, *Options*.
- All classes and domain definitions are visible to each other throughout the model.
- All new classes should be created within a sub-package of a custom model. It is advisable to model new classes within a suitable package structure. The package structure within the Cúram model is a good pattern to follow. For clarity, new class name should be prefixed with a relevant acronym or abbreviated word as discussed in the *Cúram Development Compliancy Guide*.

These are the chapters making up this part:

- Chapter 4 Audit Mappings Classes
- Chapter 5 Domain Definition Classes
- Chapter 6 Entity Classes
- Chapter 7 Extension Classes
- Chapter 8 Facade Classes
- Chapter 9 Process Classes

- Chapter 10 Struct Classes

1.6.3 Part 3 - Attribute Stereotypes

Part 3 provides reference and usage information for the attribute stereotypes used with Cúram modeling and is made up of this chapter:

- Chapter 11 Attributes

1.6.4 Part 4 - Operation Stereotypes

Part 4 provides reference and usage information for the operation stereotypes used with Cúram modeling and related topics. These are the chapters making up this part:

- Chapter 12 Operations
- Chapter 13 Entity Operations Overview
- Chapter 14 Entity Insert Operations
- Chapter 15 Entity Read Operations
- Chapter 16 Entity Update Operations
- Chapter 17 Entity Delete Operations
- Chapter 18 Entity Batch Operations
- Chapter 19 Entity Handcrafted SQL Operations

1.6.5 Part 5 - Relationship Stereotypes

Part 5 provides provides reference and usage information for the relationship stereotypes used with Cúram modeling and related topics.

Adding relationships to existing classes can be accomplished as follows:

Create a class diagram in the custom area of the model and drag the existing class(es) onto this diagram. This does not create a copy of the class, rather, it creates a reference to it. Model the relationship on this diagram as normal.

These are the chapters making up this part:

- Chapter 20 Aggregation
- Chapter 21 Assignable
- Chapter 22 Foreign Keys
- Chapter 23 Indices
- Chapter 24 Unique Indices

1.6.6 Part 6 - Other Topics

Part 6 provides reference and usage information for other modeling topics. These are the chapters making up this part:

- Chapter 25 Generated Class Hierarchy
- Chapter 26 Cúram JMS Queue Connectors
- Chapter 27 Subclassing
- Chapter 28 Application Customization

Notes

[1]UML stands for Unified Modeling Language and is an open method used to specify, visualize, construct and document the artifacts of an object-oriented software system under development.

[2]The XML produced by the Cúram Server Generator is then processed by the Cúram *Data Manager* (described in the *Cúram Server Developer's Guide*), which produces the relevant SQL scripts that are used to create the required database structure for the application.

Chapter 2

UML Overview

2.1 UML and the Input Meta-model

UML constructs, created and maintained by the user with *Rational Software Architect*, are referred to collectively as the *input meta-model* and this is used as input to the Cúram generator. It is a logical representation of the system being developed. Another way of looking at it is: it is the mechanism that developers use in order to tell the generator what to generate.

This meta-model consists of a set of packages, which in turn contain *class* representations, potentially containing *attributes* and *operations*, which have relationships with one another. Classes in the input meta-model result in various generated *Java* classes and in some cases tables and indices in generated DDL[1].

2.2 Overview of the Architecture Layers

The Cúram architecture is conceptually divided into three layers, as follows.

2.2.1 Remote Interface Layer

The *Remote Interface Layer* presents an interface to business functions that can be used by a client program. It also interacts with third-party middleware components to ensure consistency and atomicity of the transactions that execute in business functions.

2.2.2 Business Object Layer

The *Business Object Layer* implements all of the server's business functionality. As such, this layer contains the business application's “smarts”. Within this layer, *Business Process Objects* (BPOs) represent the basic business entities modeled by the server application. BPOs implement the business logic

of a Cúram server application. Typically these are responsible for manipulating Entity Objects in a business-specific way. This is where most of the development effort is (or should be) concentrated in business application development. For more information about BPOs, see Chapter 9, *Process Classes*.

2.2.3 Data Access Layer

The *Data Access Layer* is responsible for all interactions with the back end Relational Database Management System (RDBMS). For more information about Entities, see Chapter 6, *Entity Classes*.

2.3 Stereotypes

Stereotypes are a UML concept used to further describe the various aspects of the Cúram application model. In UML a stereotype is a string expression used to assign a classification to an object.

In general, stereotypes affect the behavior of the generator and thus determine its output. For example, an entity class is identified by having a stereotype of `<<entity>>` and consequently will have DDL and data-access code produced by the generator.

The supported stereotypes are as shown in the following sections.

2.3.1 Class Stereotypes

The following table lists the class stereotypes with a short description and reference to where they are described in more detail.

Stereotype	Description	Reference
audit_mappings	An audit mappings class enables additional fields to be defined in the database for auditing purposes.	Chapter 4, *Audit Mappings Classes*
domain_definition	A domain definition is a meta-model class which defines a datatype.	Chapter 5, *Domain Definition Classes*
entity	An entity class encapsulates data-maintenance functionality on a database table.	Chapter 6, *Entity Classes*
extension	Extension classes are intended to be used to change the `Audit Fields` or `Last Updated Field` options	Chapter 7, *Extension Classes*

Stereotype	Description	Reference
	of an <<entity>> or <<struct>> class.	
facade	Facade classes are used to create client-visible operations. They provide a simplified interface to a larger body of code, such as a class.	Chapter 8, *Facade Classes*
listrdo	ListRDO classes are simply an aggregation (list) of RDO classes.	*Cúram Rules Codification Guide*
loader	Loader classes are specified for rules data items in the model (<<rdo>> & <<listrdo>>).	*Cúram Rules Codification Guide*
process	A process class encapsulates a business process.	Chapter 9, *Process Classes*
rdo	RDO (Rules Data Object) classes are used to contain data used by the Cúram rules engine.	*Cúram Rules Codification Guide*
struct	A struct class is a metamodel representation of a *Java* class containing a collection of fields.	Chapter 10, *Struct Classes*
webservice	A WebService class represents an inbound legacy web service.	*Cúram Web Services Guide*
wsinbound	A WS Inbound class represents an inbound web service.	*Cúram Web Services Guide*

Table 2.1 Cúram Class Stereotypes

2.3.2 Attribute Stereotypes

The following table lists the attribute stereotypes with a short description and reference to where they are described in more detail.

Stereotype	Description	Reference
audit_mappings	An audit field entry on the Audit Mappings Class.	Chapter 4, *Audit Mappings Classes*
dataitem	A attribute on a RDO or ListRDO class.	*Cúram Rules Codification Guide*

Stereotype	Description	Reference
default	A public attribute or field in a struct class.	Chapter 10, *Struct Classes*
details	An attribute or field which is part of an entity but not part of the entity key.	Chapter 6, *Entity Classes*
key	An attribute or field which is part of an entity's key.	Chapter 6, *Entity Classes*

Table 2.2 Cúram Attribute Stereotypes

2.3.3 Operation Stereotypes

The following table lists the operation stereotypes with a short description and reference to where they are described in more detail.

Stereotype	Description	Reference
batch	Process by the Batch Launcher via generated wrapper code	Section 8.3.2, *batch* & Section 9.4.2, *batch*
batchinsert	For inserting large amounts of data via batch	Section 18.2, *BatchInsert*
batchmodify	For modifying large amounts of data via batch	Section 18.3, *BatchModify*
default	Standard non-database operation	Section 8.3.1, *default* & Section 9.4.1, *default*
insert	Standard database insert	Section 14.2, *Standard Insert*
modify	Standard database update	Section 16.2, *Standard Modify*
nkmodify	Non-key database update	Section 16.4, *Non-key Modify*
nkread	Non-key database read	Section 15.6, *Non-key Read*
nkreadmulti	Non-key database read	Section 15.7, *Non-key Readmulti*
nkremove	Non-key database delete	Section 17.4, *Non-key Remove*
ns	Database operation for handcrafted SQL	Section 19.2, *Non-standard*
nsinsert	Non-standard database insert	Section 14.3, *Non-standard Insert (Generated SQL)*

Stereotype	Description	Reference
nsmodify	Non-standard database update	Section 16.3, *Non-standard Modify (Generated SQL)*
nsmulti	Database operation for handcrafted SQL	Section 19.3, *Non-standard multi*
nsread	Non-standard database read	Section 15.4, *Non-standard Read (Generated SQL)*
nsreadmulti	Non-standard database read	Section 15.5, *Non-standard Readmulti (Generated SQL)*
nsremove	Non-standard database delete	Section 17.3, *Non-standard Remove (Generated SQL)*
qconnector	For connecting to external JMS	Section 8.3.4, *qconnector* & Section 9.4.4, *qconnector*
read	Standard database read	Section 15.2, *Standard Read*
readmulti	Standard database read	Section 15.3, *Standard Readmulti*
remove	Standard database delete	Section 17.2, *Standard Remove*
wmdpactivity	Deferred processing	Section 8.3.3, *wmdpactivity* & Section 9.4.3, *wmdpactivity*

Table 2.3 Cúram Operation Stereotypes

2.3.4 Relationship Stereotypes

The following table lists the attribute stereotypes with a short description and reference to where they are described in more detail.

Stereotype	Description	Reference
aggregation	The ability to embed or nest instance(s) of one type of class within another type of class	Chapter 20, *Aggregation*
assignable	An assignable relationship provides the ability to map differing or exclude fields for an assign func-	Chapter 21, *Assignable*

Stereotype	Description	Reference
	tion.	
extension	The link between an extension class and target class.	Chapter 7, *Extension Classes*
foreignkey	A modeled description of a database foreign key.	Chapter 22, *Foreign Keys*
index	A modeled description of a database index.	Chapter 23, *Indices*
uniqueindex	A modeled description of a database unique index.	Chapter 24, *Unique Indices*

Table 2.4 Cúram Relationship Stereotypes

2.4 Data types

The input meta-model supports a number of data types that provide abstraction for the developer from the different underlying data types used by the database, middleware and *Java* layers. These data types can be used to define attributes, arguments and return values in a platform and database neutral way, and the SDEJ will take care of mapping them to the appropriate data type in each layer of the application.

Type	Description
SVR_BLOB	Used for holding binary data. Corresponds to class `curam.util.type.Blob`. Requires a size qualifier although this is only actually used if the field is used on a database table. Fields of type `SVR_BLOB` may be null on the database.
SVR_BOOLEAN	Used for holding binary values. Corresponds to the primitive *Java* type `boolean`. Is stored as a single character field on the database where `0` = false and `1` = true. Fields of type `SVR_BOOLEAN` cannot be null on the database.
SVR_CHAR	Used for holding single character values. Note that this data type cannot be used to hold strings or arrays of characters and therefore does not take a size qualifier. Corresponds to the primitive *Java* type `char`.

Type	Description
	Fields of type SVR_CHAR cannot be null on the database.
SVR_DATE	Used for holding date values with a resolution of one day. Corresponds to class curam.util.type.Date. Fields of type SVR_DATE can be stored as null on the database.
SVR_DATETIME	Used for holding date and time values with a resolution of one second. Corresponds to class curam.util.type.Date. Fields of type SVR_DATETIME can be stored as null on the database.
SVR_DOUBLE	Used for holding floating point numbers. Corresponds to the primitive *Java* type double. Fields of type SVR_DOUBLE cannot be null on the database.
SVR_FLOAT	Used for holding floating point numbers. Used for holding floating point numbers. Corresponds to the primitive *Java* type float. Fields of type SVR_FLOAT cannot be null on the database.
SVR_INT8	An eight bit integer. Corresponds to the primitive *Java* type byte. Fields of type SVR_INT8 cannot be null on the database.
SVR_INT16	A sixteen bit integer. Corresponds to the primitive *Java* type short. Fields of type SVR_INT16 cannot be null on the database.
SVR_INT32	A thirty-two bit integer. Corresponds to the primitive *Java* type int. Fields of type SVR_INT32 cannot be null on the database.
SVR_INT64	A sixty-four bit integer. Corresponds to the primitive *Java* type long. Fields of type SVR_INT64 may be null on the data-

Type	Description
	base.
SVR_MONEY	A fixed point numeric value with two decimal places used for holding currency values. Corresponds to the primitive *Java* type `curam.util.type.Money`. Fields of type `SVR_MONEY` cannot be null on the database.
SVR_STRING	Used for holding string values. Corresponds to the *Java* class `java.lang.String`. A `SVR_STRING` may optionally have a length qualifier. A `SVR_STRING` without a length qualifier is a `SVR_UNBOUNDED_STRING`. Strings stored on the database must have a length qualifier to enable a maximum size to be specified for the database column. A `SVR_STRING` can be stored on the database as either `CHAR`, `VARCHAR` or `CLOB` depending on its size and the type of database. For more information about storage options for strings, see Section 5.1.3, *Storage Options for String Domains*. Fields of type `SVR_STRING` may be null on the database.
SVR_UNBOUNDED_STRING	Used for holding string values for which a maximum length need not be specified. Corresponds to the *Java* class `java.lang.String`. `SVR_UNBOUNDED_STRING` is the only Cúram data type which cannot be used by an attribute of an `<<entity>>` class. This is because this data type does not allow the developer to specify its maximum size and therefore cannot be used to define a database column. To define a string field on an `<<entity>>` you must use `SVR_STRING` with a length qualifier.

Table 2.5 Cúram Data Types

Notes

[1]DDL means Database Definition Language. It is an SQL language subset enabling the structure and instances of a database to be defined in a human and machine-readable form.

Chapter 3

Packages

3.1 Overview

The package structure in the UML meta-model does not affect any of the generated outputs. The hierarchy of the meta-model is effectively "flattened" during the build process.

The one area where the structure of the hierarchy is significant is that options, which can be specified at package level, will apply to all classes and other packages within that package. However, any option can be overridden in any of the sub-packages by setting the option at that level to its new value.

3.2 Options

3.2.1 CODE_PACKAGE

It is possible for two or more process or struct classes in the model to have the same name. Equally named classes are distinguished (on the server side only) by their `CODE_PACKAGE` value which may be specified for one of its containing packages.

The `CODE_PACKAGE` option, when specified, affects struct, entity, facade and process classes within that package and in the packages contained within that package. Applying the `CODE_PACKAGE` option to a class has the effect of moving that class into a package *within* the default package, `curam`, and including any of the package's parent `CODE_PACKAGE` options. The following example outlines how this works:

For example, the UML meta-model class `MyProcess` in the model causes the following *Java* classes to be created:

- `<ProjectPackage>.intf.MyProcess`

- <ProjectPackage>.base.MyProcess
- <ProjectPackage>.fact.MyProcessFactory

and the developer must implement:

- <ProjectPackage>.impl.MyProcess

If the developer wishes to create another class named MyProcess, they can do so provided that they create the class within a package for which a different CODE_PACKAGE option has been specified. This is to ensure that the corresponding *Java* classes can be stored in separate locations on disk.

The developer specifies the following option for the package containing the MyProcess class (this must be manually typed into the documentation for the package in the UML meta-model):

- CODE_PACKAGE=custom

In this instance the following classes and interfaces will result:

- <ProjectPackage>.custom.intf.MyProcess
- <ProjectPackage>.custom.base.MyProcess
- <ProjectPackage>.custom.fact.MyProcessFactory

and the developer must implement:

- <ProjectPackage>.custom.impl.MyProcess

Rules for the **CODE_PACKAGE** Feature

- CODE_PACKAGE values must be valid *Java* identifiers.
- Setting the CODE_PACKAGE option for a package recursively affects sub-packages and process, facade, entity and struct classes within the package.
- Specifying a CODE_PACKAGE value within a package whose parent has specified a CODE_PACKAGE will override the value specified by the parent rather than append to it.

  For example:

  - Package A contains package B
  - Package A specifies CODE_PACKAGE=cp1
  - Package B specifies CODE_PACKAGE=cp2

  Then:

  - The effective code package of classes in package A is cp1

    - The effective code package of classes in package `B` is `cp2` (*Not `cp1.cp2`*).

- A `CODE_PACKAGE` setting of `.` (dot) or `$` is interpreted as blank. (This is because a literal blank is ignored by the generator and therefore cannot be used to override a non-blank setting.)
- Multiple level code packages may be specified using a similar syntax to *Java* packages whereby each level is delimited by a dot. For example, the following code package setting represents three levels of *Java* packages:

  ```
  CODE_PACKAGE=cp1.cp2.cp3
  ```

- The `CODE_PACKAGE` option allows multiple struct and process classes to have the same name, however only one instance of each facade class name may exist. Cúram clients currently cannot distinguish between multiple facade classes with the same name, regardless of their `CODE_PACKAGE` setting.
- The behavior of the `CODE_PACKAGE` option with entity classes is the same as that of process and struct classes in that the resulting generated interface and struct classes are produced in different packages. However, entity class names must still be unique throughout the application regardless of the `CODE_PACKAGE` option setting. This is due to the fact that all entities correspond to tables in the single underlying database.
- Generated list wrapper structs (triggered by the existence of readmulti operations) are produced in the same code package as the structs that they wrap. Note that this will not necessarily be the same code package as the operation which caused their creation.

Chapter 4

Audit Mappings Classes

4.1 Overview

Audit Fields are fields which can be added to database tables to contain extra information about the modification history of each record for auditing purposes.

Audit fields are only available on entity and struct classes and are updated only by certain entity operations.

The information specified in audit fields can be specified by the developer. Typically, the audit fields should include the following:

- Creation time;
- Modification time;
- Program ID;
- User ID.

Audit fields consist of all the attributes of a special class in the input metamodel called `AuditMappings`. A field corresponding to each attribute of this class can then be automatically added to the database table, and also to all the standard details structs for the entity.

4.2 Rules

The following rules apply to the `AuditMappings` class:

- The stereotype must be `<<audit_mappings>>`.
- The attributes of the class must be valid domain definitions.
- The class must be “flat”, i.e. it cannot aggregate any other classes.

Audit mappings are made available to an application by adding a class named `AuditMappings` with a `<<audit_mappings>>` stereotype to the model. Individual entity classes can then enable audit mappings by setting the `Audit Fields` option.

If the meta-model contains an `AuditMappings` class then a *Java* implementation class for it must be provided in the `impl` package.

Note

If this implementation class is not present, the server application cannot be compiled. In this situation the developer should either:

- delete the `AuditMappings` class from the model
- explicitly disable audit mappings completely by specifying the generator switch `-noauditmappings`.

The following rules apply to the `AuditMappings` implementation class:

- it must contain the same fields as defined in the meta-model (i.e. they must have the same name and data type.)
- these fields must be public
- it need not inherit from any other class
- it may optionally contain the following method:

  ```
  public void set(final boolean isInsert, final boolean isModify)
  ```

  This is a call back method which is called whenever necessary (i.e. during inserts and modifies) by the data access layer and should be used to populate the fields of the `AuditMappings` class. The two boolean parameters indicate whether the database operation is an insert or modify, respectively.
- it may optionally contain a public void method named `set` which takes no parameters. This method will be called by the data-access layer whenever it needs the fields to be updated. (In fact any public method whose name starts with `set` and which takes no parameters will be called in arbitrary order, but it is not recommended to use multiple setter methods and support for doing so will be discontinued in future.)

If the details struct contains any of the audit mapping fields, then these are updated in the struct automatically during the operation and are included in the update or insert.

For audit mapping fields, which are not present in the details struct, the corresponding field will still be updated on the database, i.e. it is not necessary to include the audit mapping fields in the details struct to get them updated on the database. Note however that such fields are not included in table level auditing.

4.3 Outputs

Switching on auditing for an entity has the following effect:

- Fields are automatically added to the entity and to the generated standard details struct for the entity.
- Infrastructure data-access code automatically makes calls to the AuditMappings class to populate its fields whenever audit fields are being updated on the database.

The following operation stereotypes cause audit information to be set:

- <<modify>>;
- <<nsmodify>>;
- <<insert>>;
- <<nsinsert>>;
- <<nkmodify>>;
- <<batchinsert>>;
- <<batchmodify>>.

4.4 Options

Two options are available for attributes of the AuditMappings class in the model:

- Exclude from insert
- Exclude from modify

If Exclude from modify is set for an audit mappings field, then the value of this field will not be changed by a <<modify>> / <<nsmodify>> / <<nkmodify>> operation. i.e. the field will be set when a record is inserted, and will never be changed by subsequent updates. Similarly if Exclude from insert is set then the value of the field will not be set by a <<insert>> / <<nsinsert>> operation but will be changed by any subsequent updates. The default value for each of these options is false.

Note that it is not possible to cause audit mapping fields to be excluded from operations of stereotype <<ns>>. Handcrafted SQL in these operations can still be used to access audit mapping fields directly.

Note

If your audit mappings include a time stamp then you should popu-

late this field with the value returned by `TransactionInfo.getProgramTimeStamp()`. This will ensure that all audit mapping-enabled tables modified during the transaction will have the same time stamp value even though the tables will not have been written to at the exact same time.

Chapter 5

Domain Definition Classes

5.1 Overview

In relational database terminology, a domain defines the range of values allowed for an attribute of an entity. *IBM Cúram Social Program Management* uses domain definitions in a similar way. Domains are datatype definitions which resolve to either a primitive datatype or another domain. Equivalent primitive types are supported across client, middleware, server and database components of a Cúram application:

Cúram Architecture Layer	Datatypes
Server Remote Interface Layer	*Java* datatypes
Server Business Object Layer	*Java* datatypes
Server Data Access Layer	*Java* datatypes
Database	Database datatypes

Table 5.1 Domain primitive types at different levels of a Cúram application

By working with domains, rather than primitive types, developers are protected from having to worry about different representations of data in the various application layers. For this reason, entity and structure attributes must be defined in terms of a previously defined domain - it is not possible to use primitive datatypes directly.

Validations on each domain type are also allowed to be defined in the client application. A specific validation can then be executed for all attributes defined in terms of a given domain type, before transactions are invoked on the server. This client-side pre-flight validation gives the user feedback on basic datatype validation without having to call the server, resulting in lower network overhead because of the reduced number of failed transactions.

5.1.1 Defining a Domain Hierarchy

Another advantage of using domains is that it allows changes to the datatypes of related attributes to be effected simply by changing a domain definition. Say, for example, that a particular type of reference number changes from a 10-digit to a 12-digit number. The reference number probably appears as an attribute in many different entities and structures. As long as these attributes have been defined in terms of a common domain definition, they can all be changed together by modifying the domain definition (obviously, there will also be database impact, etc. to consider).

By defining domains in terms of other domains, it is possible to set up a hierarchy of related domain definitions. For instance, consider the following entity classes and their domain-specified attributes:

- <<entity>> class: Customer

Attribute	Domain
<<details>> address_1	CUSTOMER_ADDRESS_LINE
<<details>> address_2	CUSTOMER_ADDRESS_LINE
<<details>> address_3	CUSTOMER_ADDRESS_LINE

- <<entity>> class: Employer

Attribute	Domain
<<details>> address_1	EMPLOYER_ADDRESS_LINE
<<details>> address_2	EMPLOYER_ADDRESS_LINE
<<details>> address_3	EMPLOYER_ADDRESS_LINE

In the above tables, the address attributes of the Customer and Employer entities are defined in terms of CUSTOMER_ADDRESS_LINE and EMPLOYER_ADDRESS_LINE respectively. Both of these domains are in turn defined in terms of the ADDRESS_LINE domain. All of the domains ultimately unwind to a 30-character string primitive datatype.

The following rules of thumb should be followed when defining attributes:

- attributes whose types must be able to vary independently of each other should be defined in terms of different domains;
- attributes that should always have the same types should be defined in terms of the same domain;
- attributes that initially have the same type, but might in the future vary independently should be defined in terms of related domains.

Thus, in the example, it is possible to change the ADDRESS_LINE domain in order to change the types of all entity address line attributes, but the Cus-

tomer and Employer address line attributes can also be varied independently. Given different design decisions, the entity address line attributes might just have been defined in terms of ADDRESS_LINE (on the assumption that all address lines will always have the same type), or each address line might have had a separate domain definition (on the assumption that address lines 1, 2, and 3 might not always be the same size).

5.1.2 Proper Use of Domains

Getting the granularity of domain definitions right is important - too few separate definitions might make it difficult to change the datatypes of some attributes without impacting others; too many definitions make it difficult to follow which attributes are related to which others. Remember that the granularity also determines at what level validations on attributes can be implemented in the Cúram web client.

A design in which every attribute has a different associated domain is probably wrong. At the very least, attributes which are foreign keys should share their domain definitions with the original key.

In general, an analysis of the types of data your application uses early in the design stage is probably the best approach to coming up with a sensible domain hierarchy. You should also decide at this point which domains will require special client-side validations to be constructed.

5.1.3 Storage Options for String Domains

There are three categories of database storage for string: small, medium and large, corresponding to the maximum sizes of the CHAR, VARCHAR and CLOB data types in the database. By default, the Cúram generator will place each string domain definition into the smallest possible category based on its size.

For example, in *IBM® DB2®* the maximum size of a CHAR column is 254 and the maximum size of a VARCHAR column is 32768, so a SVR_STRING of up to 254 will be categorized 'small', a SVR_STRING from 255 to 32768 will be categorized “medium”, and larger strings will be categorized as “large” and are stored as a CLOB.

The Storage Type option allows developers to specify that a string be treated as a small/medium/large regardless of the size of a string. For example, in *DB2*, this enables developers to use VARCHAR or CLOB instead of CHAR, or CLOB instead of VARCHAR, if necessary.[1]

The decision to override the default selection of small/medium/large - i.e. CHAR/VARCHAR/CLOB - is a database tuning exercise which should involve the developer and DBA, and can be quite complex. For example CHAR can be more performant than VARCHAR but uses more space. And while VARCHAR can save space, it can lead to row migration if not tuned correctly. Database tuning is the responsibility of the DBA and is not covered by this document.

This option is applicable to all domain definitions whose eventual type is a SVR_STRING. Specifying this option on a domain definition will affect that domain definition and all domain definitions derived from it - unless it is overridden in one of the derived domain definitions.

For example, consider the following domain definitions:

- PHONE_NUMBER is a SVR_STRING<32>.

  This domain definition does not have a storage_type option specified so the size 32 means that this domain definition will have a default storage_type of small i.e. it will be stored as CHAR on the database.

- BUSINESS_PHONE_NUMBER is a PHONE_NUMBER

  This domain definition specifies a storage_type of medium. So instead of CHAR it will be stored as VARCHAR on the database.

- ALTERNATE_BUSINESS_PHONE_NUMBER is a BUSINESS_PHONE_NUMBER

  This domain definition does not have a storage_type option specified so it inherits the value storage_type specified in BUSINESS_PHONE_NUMBER. Therefore it will be stored as VARCHAR on the database

So while the underlying business meaning of the above three domain definitions is the same - all are phone numbers - they can be stored differently on the database as appropriate.

Note

For database operations on entities in which the parameters are specified by the developer (i.e. <<nsread>>, <<ns>>, etc.) it is necessary to ensure the domain definitions used in the parameter structs are the same or at least compatible with those in the entity. This is because the type of the domain definition in the parameter struct determines the type of host variable which gets produced in the generated data-access-layer. For example, in *DB2*, this could mean that a CLOB gets read into a CHAR host variable, or a VARCHAR gets read into a CLOB host variable, etc. The combinations permitted are different depending on the target database type. Caution is advised whenever custom parameters are specified in this way.

Note

For *DB2* and *IBM® DB2® for z/OS®* the allocation of strings to CHAR, VARCHAR and CLOB in the database may be impacted by the setting of the Multibyte_Expansion_Factor storage option and related build-time settings. See Section 5.2.11, *Multibyte Expansion Factor* and the *Cúram Server Developer's Guide* for more information.

5.2 Options

The following are the options allowed for Domain Definitions:

5.2.1 Code Table Name

This specifies the name of the code table which contains valid entries for this Domain Definition. If the domain definition represents a hierarchy of code tables, the name of the lowest code table in the hierarchy should be specified as the code table name.

For fields for which a code table has been specified, the client application will display a drop-down list of valid values for the field if it is editable, or the code table translation for the field if it is read-only. In the case of a code table hierarchy, if the code table field is editable, n-levels of drop-down lists are displayed, where n is the number of code tables in the hierarchy. Only the first level is populated and a selection must be made to populate the next level in the hierarchy. For a read-only field where the code table is a hierarchy, the translation for the lowest level code table only is displayed.

This option is only valid for Domain Definitions which have been defined in terms of one which has the `Code Table Root` option set to `yes`.

5.2.2 Code Table Root

Specifies whether the current Domain Definition is the root of a hierarchy of code table Domain Definitions. If this is set to `yes`, then all Domain Definitions which use this one (i.e. they are defined in terms of it) must specify the `Code Table Name` option. If the developer forgets to specify the `Code Table Name` option, an error will be displayed by the generator.

Since this Domain Definition will be used to hold code table codes, its type should match that of a Cúram code table code, i.e. `SVR_STRING<10>`.

For more information see Section 5.2.1, *Code Table Name* above.

5.2.3 Compress Embedded Spaces

Implemented in the Cúram client application.

It specifies that any *extra* whitespace [2] (not all whitespace) embedded in the string, and that all leading and trailing whitespace is removed before being sent to the server.

Extra whitespace consists of a run of whitespace characters immediately *after* another whitespace character. This means that each run or sequence of whitespace characters is deleted except for the first whitespace character of the run. For example, a pair of words separated by three spaces will be converted to the pair of words separated by one space.

Note that in cases where the first whitespace character is not a space, the

results may not be as expected. For example, a pair of words separated by carriage-return, line-feed, space, space will be converted to the pair of words separated by the carriage-return character.

Note also that if this feature is used on multiple line text fields it will remove indentation.

Note

Switching on this option also causes leading and trailing whitespace to be trimmed from the string, regardless of the `Remove Leading Spaces` and `Remove Trailing Spaces` option settings.

5.2.4 Convert to Uppercase

Implemented in the Cúram client application.

It specifies that the contents of this string field be converted to uppercase before being sent to the server.

5.2.5 Custom Validation Function Name

Domain Definition validations implemented in the client infrastructure include a custom validation type which corresponds to a developer-supplied function for performing validations on data entered by users via the client interface.

This option allows the developer to specify the name of this function which associates it with the application UML model. The value of the option should be simply the name of a function (just function, not class + function, since the class name is defaulted in the client code). It must also be a valid *Java* identifier.

Note

This feature has been deprecated, please see the “Custom Data Conversion and Sorting” chapter of the Web Client Reference Manual for information on the new domain plug-in system.

5.2.6 Default

Implemented in the Cúram client application.

It specifies that this field will contain a default value after it is displayed.

5.2.7 Maximum Size

Implemented in the Cúram client application and database DDL.

It specifies a maximum number of characters which can be entered to this field before it can be sent to the server and forms the field storage size on the database.

5.2.8 Maximum Value

Implemented in the Cúram client application.

It specifies a maximum permitted numeric value which must be entered into this field before it can be sent to the server.

5.2.9 Minimum Size

Implemented in the Cúram client application.

It specifies a minimum number of characters which must be entered to this field before it can be sent to the server.

5.2.10 Minimum Value

Implemented in the Cúram client application.

It specifies a minimum permitted numeric value which must be entered into this field before it can be sent to the server.

5.2.11 Multibyte Expansion Factor

Implemented in the *Data Manager* for *DB2* and *DB2 for z/OS* only.

For string domains it specifies an expansion factor (float from 1.0 to 4.0) to be applied when multibyte character set (MBCS) data will be used with *DB2* or *DB2 for z/OS*. It overrides the global build-time property (`curam.db.multibyte.expansion.default.factor`) and is only necessary in order to deviate from the global setting (e.g. a particular domain is causing a *DB2* limit to be exceeded). A setting of 1.0 effectively turns off expansion for this domain. You might choose to set this option for domains where you know the contents will never contain localized data; e.g. they are constrained to programmatically-defined Western characters and can't be input via a client. The same option set for a string entity attribute can override this domain setting (see Section 11.3.2, *Multibyte Expansion Factor* for more information). This option is ignored if the feature is turned off (`curam.db.multibyte.expansion` set to `false` at build time; see the *Cúram Server Developer's Guide* for more information).

5.2.12 Pattern Match

Implemented in the Cúram client application.

It specifies a regular expression that the string value must match before it can be sent to the server. The regular expression must match the whole string, not just a portion of it. The regular expression syntax is the standard *Java* regular expression syntax used in *Java* 1.5. Full details on the supported syntax for these regular expressions can be found in the JavaDoc documentation for the `java.util.regex.Pattern` class supplied with

your *Java* SDK.

5.2.13 Remove Leading Spaces

Implemented in the Cúram client application.

It specifies that any leading spaces be stripped off the string before it is sent to the server.

5.2.14 Remove Trailing Spaces

Implemented in the Cúram client application.

Specifies that any trailing spaces be stripped off the string before it is sent to the server.

5.2.15 Storage Type

It allows the developer to specify what type of string storage datatype to use for this domain definition on the database. See Section 5.1.3, *Storage Options for String Domains* for more information. This option is only relevant for string domain definitions for which a length has been specified.

5.3 Overriding a Domain Definition

The SDEJ provides the facility to override existing domain definitions without modifying the original domain definition. This is desirable in situations where the original domain definition is provided by a third party and should not be modified locally.

Suggested uses:

- Change the maximum size of a string field.
- Change the `Storage Type` of a domain definition.

5.3.1 How to use Domain Definition Overrides

A domain definition is overridden by creating a new domain definition with the same name prefixed by an asterisk. For example, the domain definition, `PERSON_NAME`, would be overridden by creating a domain definition named `*PERSON_NAME`. At build time the overridden version is used instead of the original version, complete with its own data type and options.

5.3.2 Considerations / Limitations

- It is important to be aware that overriding a domain definition affects all usages of the original domain definition. It is the responsibility of the developer to ensure that pre-existing functionality is not broken by over-

riding domain definitions. Specifically attempting to change the Type of the domain definition, the Code Table Name or the Code Table Root is discouraged.

5.3.3 Usage Rules

- A domain definition may be overridden by only one override.
- A domain definition override cannot be overridden. For example, if PERSON_NAME is overridden by *PERSON_NAME, it is *not* permitted to further override *PERSON_NAME with **PERSON_NAME.
- It is not possible to create overrides for domain definitions which do not exist. For example, if there is a domain definition override named *PERSON_NAME then the model must contain a domain named PERSON_NAME.
- Domain definition overrides cannot be used as attributes of structs or entities, i.e. attributes cannot use domain definitions whose names begin with an asterisk.

Notes

[1]Note that if the developer specifies a Storage Type which is too small for the actual size of the string, the next smallest category will be used. For example, (in *DB2*), if a developer specifies a `Storage Type` of `small` for a `SVR_STRING<1000>` the generator will still treat this as a `medium` string since the maximum size of a `small` string (i.e. `CHAR`) in *DB2* is 254.
[2]A whitespace character consists of any character for which `java.lang.Character.isWhitespace(char)` returns `true` . Such characters include the space character, the tab character and the line-feed character.

Chapter 6

Entity Classes

6.1 Overview

Entity classes have a stereotype of `<<entity>>`.

An entity is a collection of fields and their associated database operations. Entity classes are the fundamental building blocks of systems developed with the *IBM Cúram Social Program Management* framework. They correspond to database tables and are the type of construct for which the Cúram generator gives the most support in terms of automatic code generation.

An entity class is essentially an object wrapper for a database table. The attributes of an entity are transformed to columns on the database table. Entities can have various data maintenance operations such as reads, inserts, modifies, removes, readmultis (which read multiple records from a table based on a partial key), etc. Standard operations (e.g. read, insert, etc.) operate on one database table by default. For example, in a banking system you could have an Account entity class whose operations would include insert, read, update, etc.

Entities are allowed to have attributes, operations, dependencies, inherits relations, and aggregations. Each of these constructs has a set of rules associated with it, which are detailed below.

For more information, see Section 6.4, *Operations*

6.2 Rules

- Entities must have at least one attribute *unless* the entity is a subclass of another entity, in which case it must have no attributes.
- Entities are not allowed to aggregate other classes.

6.3 Attributes

Entity attributes correspond to columns with the same name on their associated database table.

Attributes are not contained in the generated BOL or RIL. This is because Cúram interface objects are stateless and atomic. Instead these attributes are contained within generated standard key and details structs (see Section 10.3, *Outputs*).

The stereotype of an entity attribute cannot be blank. It must be one of the following:

6.3.1 Details

The attribute is included as a column on the database table and in the standard details struct for the entity. For more information, see Section 6.5.2, *Standard Details Structs*.

6.3.2 Key

The attribute is included as a column on the database table, it forms part of the primary key, it is included in both the standard details struct and the standard key struct for the entity. For more information, see Section 6.5.1, *Standard Key Structs*.

6.4 Operations

Entity operations can be divided into two categories as determined by their stereotype:

6.4.1 Database Operations

These are operations whose stereotype is recognized by the generator. These operations are fully or partially generated by the generator and operate directly on the RDBMS table related to the entity. They include standard operations to *read*, *insert*, *update*, *delete*, together with their variants.

6.4.2 Non-database Operations

These are operations whose stereotype is not recognized by the generator. The generator generates only prototypes and skeletons for these operations, no data-access operations are generated. The body of these functions must be implemented in the BOL by the developer.

The operations available for entity classes are listed in Section 2.3.3, *Operation Stereotypes*.

6.5 Outputs

Entity classes are transformed into classes with operations and no attributes. The attributes from the entity in the input meta-model are transformed into one or more structs.

6.5.1 Standard Key Structs

Standard key structs are generated for entity classes and contain those attributes in the class whose stereotype is `<<key>>`. If no such attributes exist in the class then a standard key struct is not generated.

This struct will be used as a parameter for operations requiring a primary key. For example, reads and deletes.

Though standard key structs do not appear in the input meta-model they can be used as arguments to operations in the input meta-model. The name given to standard key structs is the name of the corresponding entity with the word `Key` appended. For example, the standard key struct for the class `Employer` would be called `EmployerKey`.

6.5.2 Standard Details Structs

Standard details structs are generated for all entity classes and contain all the attributes of the class. This struct is used as a data parameter to insert, reads and updates. Structs containing arrays of standard details structs are returned from standard readmulti operations.

Though standard details structs do not appear in the input meta-model they can be used as arguments to operations in the input meta-model.

The name given to standard details structs is the name of the corresponding entity with the word `Dtls` appended. For example, the standard details struct for the class `Employer` would be called `EmployerDtls`.

6.5.3 Standard List Structs

Standard list structs are generated for entity classes which contain one or more operations of stereotype `<<readmulti>>` or `<<nkreadmulti>>`. This struct contains a single attribute named `dtls` which is a sequence of the standard details struct for the entity.

The name for a standard list struct is the name of the standard details struct for the entity with the word `List` appended. For example, the standard details struct for the class `Employer` would be called `EmployerDtlsList`.

6.6 Options

The options available for entity classes are described in the sections below.

6.6.1 Abstract

Specifies that the class is abstract. Abstract classes are intended to be sub-classed by other classes. For more information on abstract classes and sub-classing, please see Chapter 27, *Subclassing*.

6.6.2 Allow Optimistic Locking

Only applicable for entities which are not subclasses. Optimistic locking is supported on certain database operations (see Section 12.3.14, *Optimistic Locking* in Section 12.3, *Operation Options*). In order to use optimistic locking on an entity's operation, this option must first be switched on for the class.

For more information on optimistic locking, see Section 6.7, *Concurrency Control - Optimistic Locking*.

6.6.3 Audit Fields

Only applicable for entities which are not subclasses. Extra fields can be configured to store additional information on a database table for auditing purposes. These fields are covered in more detail in Chapter 4, *Audit Mappings Classes*.

If this option is switched on, then the available pre-configured audit fields will be automatically added to this entity and its standard details struct.

6.6.4 Enable Validation

The validation operation is an exit point which gets called automatically for the purpose of validating data. This exit point will get called before the data-access layer entity operations whose stereotype is `<<insert>>` or `<<modify>>`.

For more information on exit points, see Section 6.9, *Exit Points*.

6.6.5 Last Updated Field

Only applicable to entity classes that are not subclasses.

In order to use the last updated field feature for an entity class, this option must first be switched on. This results in the addition of an extra timestamp field to the specified entity. This field gets updated with the current date and time whenever the record is written - unless the write was performed by an `<<ns>>` operation. For more information on the last updated field feature, see Section 6.11, *Last Updated Field*.

6.6.6 No Generated SQL

Switches on the `No Generated SQL` for all database operations of the entity class. Individual entity operations can override the value of this option.

For more details see Section 12.3.12, *No Generated SQL* in Section 12.3, *Operation Options*.

6.6.7 Replace Superclass

This is only relevant to entities which are subclasses.

If this option is set, then requests to create instances of the superclass will instead result in the creation of the subclass. This enables the developer to change functionality by replacing subclasses with other classes.

6.7 Concurrency Control - Optimistic Locking

Using optimistic locking for concurrency control means that more than one user can access a record at a time, but only one of those users can commit changes to that record. Once one user has modified the record, another user cannot modify it without first re-reading the latest version of the record. Thus it is optimistic in the sense that one user does not expect another to attempt to modify the same record at the same time.

The record being edited is locked for update only while the changes are being committed. This has the advantage of minimizing the time for which a lock is in place.

The disadvantage of optimistic locking is that when a user begins to edit a record, they cannot be sure that the update will succeed. An update that relies on optimistic locking will fail if another user has updated a record while the first user is still editing it.

Optimistic locking is implemented by adding an extra field to the database table. The extra field contains the version number for the record and is automatically incremented each time the record is modified. The generated DAL code checks this version number while the record is being updated, and if the version number on the database table is not the same as the version number on the original record then the update operation is aborted and an exception is thrown.

Optimistic locking is permitted only on Entity classes.

The following operation stereotypes support optimistic locking:

- `<<modify>>`;
- `<<nkmodify>>`;
- `<<nsmodify>>`.

The following operation stereotypes are affected by optimistic locking:

- <<insert>> - The version number field is automatically included in the details parameter and is automatically initialized before being written to the database;
- <<nsinsert>> - If optimistic locking is enabled on an entity class, the version number field *must* be included in the details struct by the developer and will be automatically initialized before being written to the database.

Optimistic locking is only possible for operations which modify a single database record and whose details struct includes the generated Version Number field. This means that for non-standard operations, it is up to the developer to ensure that the non-standard key parameter always identifies a single unique record and that the Version Number field is included in the details struct. For <<nkmodify>> operations, optimistic locking is only possible if the database table contains exactly one record. This field must be called versionNo and its type should be VERSION_NO. The developer must ensure that the model contains a numeric domain definition named VERSION_NO.

In order to support optimistic locking on an operation you must do two things:

- Switch on the Allow Optimistic Locking option on the entity.

  This will cause the Version Number field to be automatically added to the entity.
- Switch on the Optimistic Locking option on the operation.

  This will cause the generator to generate code in the DAL for the operation which will check and update record version numbers accordingly.

6.8 Table Level Auditing

Auditing is supported on all stereotyped entity operations except <<ns>>, <<nsmulti>>, <<batchinsert>> and <<batchmodify>>.

The information captured by table level auditing is stored in the database table AuditTrail.

Table level auditing is enabled by switching on the Database table-level auditing option for an operation. This causes the generated data-access code to record audit information for an operation.

The type of audit information recorded depends on whether optimistic locking is switched on or off for the operation. If optimistic locking is switched on, then the audit information includes the information of the new and old versions of the record, otherwise it only includes information about the SQL operation invoked.

6.8.1 Information Captured by Table-level Auditing

The following information is captured:

- *Date and time* - The date and time of the transaction.
- *User ID* - The ID of the user who invoked the transaction.
- *Table name* - The name of the database table which was modified.
- *Program name* - The FID of the function which invoked the transaction.
- *Transaction type* - Indicates whether the transaction was *online* / *batch* / *deferred* / etc.
- *Key info* - The key which was provided to this operation. Note that this may identify one or many records.
- *Details of changed data* - These details are logged in an XML format. The exact format of this XML can be seen in the JavaDoc details for the class `curam.util.audit.AuditLogInterface` in the `doc/api` directory of the SDEJ. They include the names of the all the fields referenced by the details struct, the field types, the new version of the field data and, if optimistic locking is enabled, the old version of the field data.

  If optimistic locking is switched on, then the operation is guaranteed to have only affected a single record. Therefore, the audit information includes information about the record before and after the operation. The old version of the record is re-read, the old value of each field is compared to the new value, and any field which has changed is included in the audit information, i.e. unchanged fields are filtered out.

  If optimistic locking is switched off then for performance reasons the record is not re-read during the update, so the audit information will contain only the new versions of all the fields involved in the update, not a *before-after* comparison of the record. Also, any non-optimistic updates apart from the 'modify' stereotype can potentially affect more than one record, in which case it is not possible to record a before-after comparison of the update. All the detail fields will be included regardless of whether the new value is different to the old value.

  This data can be compressed when using the default auditing handler by specifying the `curam.audit.audittrail.datacompressionthreshold` property described in the *Cúram Configuration Parameters* appendix of the *Cúram Server Developer's Guide.*
- *Operation type* - Indicates whether the operation was one of: create, read, update or delete.

6.8.2 Storage of Audit Information

By default, the audit information captured is written to the AuditTrail database table. The developer may also supply their own auditing handler by

specifying a class which implements the `curam.util.audit.AuditLogInterface` interface. For more information, see the section on Customization Settings in the *Cúram Server Developer's Guide*.

6.9 Exit Points

An exit point is a callback function written by the developer and executed at a predefined strategic point by the server.

Four types of exit point are supported:

6.9.1 Pre Data Access

This function is called before the DAL function (but after *Validate* functions).

The function is named after the method to which it belongs, prefixed with `pre`, e.g. `preread`.

6.9.2 Post Data Access

This function is called after the DAL function.

The function is named after the method to which it belongs, prefixed with `post`, e.g. `postread`.

6.9.3 Validation

This function is called before standard insert and standard update operations, and also before *Pre-data Access* functions. It provides a common place to put validation code.

The function is named `autovalidate`. Note that this exit point is enabled per entity rather than per operation.

The validation exit point always has exactly one parameter which is the standard details struct for the entity, and is declared to throw the same exceptions as stereotyped operations of the entity.

Since it is only `<<insert>>` and `<<modify>>` which are guaranteed to pass in the standard details struct, it is only these operation stereotypes which can utilize the validation exit point. Other operation stereotypes do not utilize this exit point, even if they have the standard details struct as one of their parameters.

6.9.4 On-fail

This function is called if an error occurs in the data access function.

The function is named after the method to which it belongs, prefixed with

onFail, e.g. onFailread.

Note

For non-void operations the return class is included in the arguments to this method and will always be *null*.

6.9.5 Exit Point Parameters

With the exception of *Validate* exit points whose parameters are described above, the parameters to an exit point method consist of the following:

- the parameters to the method to which the exit point belongs. (In fact if any extra parameters have been specified for a database operation in the model, this is the only place where the developer can access them.)
- the return type of the method to which the exit point belongs - if a return type is present.

  

  Limitation

  The return type parameter will not be included into the parameters of exit point methods for <<nsread>> and <<ns>> operations.

  The following approach can be used to generate the return type parameter into the parameters of exit point methods for <<nsread>> and <<ns>> operations:

  - Add an un-stereotyped method to the entity class giving it the same signature as the <<nsread>> or <<ns>> operation.
  - Set the *Post Data Access* option on your <<nsread>> or <<ns>> operation to False.
  - The implementation of your un-stereotyped operation will then call the <<nsread>> or <<ns>> operation, and will have access to its return value as required.

- for on-fail exit points, an exception class. This is the exception which was thrown from the data access layer. The exit point may handle the error or pass it on by throwing it.

6.9.6 What should exit points be used for

Exit points are intended to be used for validation or for completing a business process. For example, after modifying an invoice detail line record, the last modified date should be updated on the invoice header record.

6.9.7 What should exit points *not* be used for

It is recommended that exit points should not be used as a means of populating incomplete fields in incoming parameters. This situation should be handled by wrapping the database function in a non-database function which would take a copy of the incomplete record, fill in the missing fields and invoke the database operation.

When adding an exit point to an entity operation, ensure that this will not have any side effects for other users of the operation.

6.10 Entity Inheritance

Input meta-model Entity classes are allowed to subclass other entity classes. Typically, entity classes will be subclassed in order to add functionality (such as additional stereotyped operation) which are required for special processing of the associated database table, but which do not belong in the parent class.

For more information please see Chapter 27, *Subclassing*.

6.10.1 Rules when Using Entity Inheritance

- Entity classes are only allowed to inherit from other entity classes.
- Subclasses of entities can add any number of additional database and user-defined operations.
- Subclasses of entities cannot add attributes. This is because the underlying relational database table must not be affected by the inheritance.
- Entity subclasses do not have standard key and details generated for them, they use the standard key and details structs from the base class.

6.11 Last Updated Field

This feature is similar to the Audit Mappings feature. It is a field which can be added to database tables to contain extra information about the modification time of each record for reporting purposes.

The feature is only available for entity classes and it is updated only by certain entity operations

Switching on the last updated field functionality for an entity has the following effects:

- A field called `lastWritten` of type `SVR_DATETIME` is automatically added to the entity.
- The Cúram infrastructure automatically populates this field with the current time whenever the record is written to the database - unless the write was performed by an `<<ns>>` operation.

The following steps must be taken to avail of this feature:

- To turn on the feature for an individual entity class the Last_Updated_Field property in the *Rational Software Architect* Cúram Properties tab must be set to '1 - yes' using the supplied drop-down.
- To turn on the feature for all of the entities for a particular application, the following text must be appended to the `extra.generator.options` property in the `Bootstrap.properties` file as follows:

  ```
  extra.generator.options=-defaultoption class_lastupdatedfield=yes
  ```

  Example 6.1 extra.generator.options property in Bootstrap.properties

  ```
  extra.generator.options=-defaultoption
  class_lastupdatedfield=yes
  ```

- A new domain definition must be specified in the model as follows:
  - Domain Definition Name: LAST_UPDATED
  - Domain Definition Type: SVR_DATETIME

Invoking operations with the following stereotypes cause the *lastWritten* field to be set:

- `<<insert>>`;
- `<<nsinsert>>`;
- `<<modify>>`;
- `<<nkmodify>>`;
- `<<nsmodify>>`.

Note

Unlike the version number field utilized for the optimistic locking feature, there is no requirement to add the last written field to structures involved in non-standard insert and modify operations. If the last updated field feature has been enabled for an entity, this field is always updated for the operation stereotypes listed above by the infrastructure data access code regardless of whether the field is present in the structure being used.

Chapter 7

Extension Classes

7.1 Overview

An extension class allows the developer to specify options for a target class without modifying the meta-model definition of the target class. Each extension class should be linked to one target class. At build time the contents of an extension class are effectively super-imposed on its target class.

7.2 How to use Extension Classes

To extend an existing class, create a new class of stereotype `<<Extension>>`; see *Working with the Cúram Model in Rational Software Architect* for more information on using and modeling with *Rational Software Architect*.

Options can be added to the extension class in the same way as for other classes. When any of these are added to an extension class they have the effect of adding (if not already existing) or modifying (if already existing) the same named option on the target class.

When creating an Extension class in *Rational Software Architect*, keep in mind that since it can apply to different class types. You must make sure the settings for the extension class are compatible with the class you are extending.

The extension class is linked to its target class by adding an relationship of stereotype `<<extension>>` between the two classes. The new class should be created within a custom sub-package of your model.

7.3 When to use Extension Classes

Extension Classes should only be used for the following purpose:

- To switch on the `Last Updated Field` option on an entity.
- To switch on the `Audit Fields` option on an entity or struct.

Note

The *Rational Software Architect* user interface allows you to specify additional information such as attributes and options for `<<extension>>` classes but only the above two options should be included in the class. Other changes are not compliant.

7.4 Considerations / Limitations

- Storage of relationships. When a relationship is created between two classes in *Rational Software Architect*, it should be noted that the relationship is not stored within either of the actual classes but rather as a free standing object in a package. (Usually the relationship is stored in the package containing the diagram on which it was drawn but this is not guaranteed.) The developer should ensure that the relationship is stored in a location where it will not be lost or overwritten during an upgrade. Inheritance relationships are always stored within the subclass so there is no risk of inadvertently losing these.

7.5 Usage Rules

- An extension class may only be applied to one target class.
- A class may be extended by multiple extension classes.
- Extensions may be applied to classes of stereotype `<<entity>>`, `<<struct>>`.

Chapter 8

Facade Classes

8.1 Overview

A *facade* class is defined as a class which encapsulates a business process that is visible to the client. They form the *Business Object Layer* (BOL) of the application. It is a collection of operations. Facade classes do not have data maintenance operations, or indeed any relationship with database tables. Instead they manipulate other entity and process classes in order to implement a business process.

Facade classes have a stereotype of `<<facade>>`.

8.2 Rules

- Facade classes must have a stereotype of `<<facade>>`.
- Facade classes cannot have aggregations to any other classes.
- Facade classes can only inherit from other facade classes - not from entity or process classes.
- Facade classes cannot have attributes.
- Facade classes cannot have the same name. See Section 3.2, *Options* for more details.

8.3 Operations

Within facade classes there are four operations supported:

8.3.1 default

The `<<default>>` stereotype offers a standard or plain operation.

8.3.2 batch

For operations of stereotype <<batch>>, the Cúram generator will produce the necessary source code wrappers to build a batch wrapper program which will enable this operation to be run by the Batch Launcher (see the *Cúram Batch Processing Guide* for more details).

The rules when defining batch operations are:

- Batch operations cannot have more than one parameter.
- Parameters to batch operations must be structs.
- A facade class cannot have more than one <<batch>> operation.

8.3.3 wmdpactivity

A method of a facade class can be designated as a deferred processing method by setting its stereotype to << wmdpactivity>>.

For more information please consult the *Cúram Server Developer's Guide*.

8.3.4 qconnector

For operations of stereotype <<qconnector>>, the generator will produce the necessary source code to connect to a JMS provider (e.g. *IBM® MQSeries®*). For more information please see Chapter 26, *Cúram JMS Queue Connectors*.

8.4 Options

8.4.1 Abstract

Specifies that the class is abstract. Abstract classes are intended to be subclassed by other classes. For more information on abstract classes and subclassing, please see Chapter 27, *Subclassing*.

8.4.2 Generate Facade Bean

This causes a stateless session bean to be generated for this class. This bean class can be used to allow your server to be accessed by other systems or by message driven beans.

8.4.3 Replace Superclass

This is only relevant if this facade class has been subclassed from another class. For more information on subclassing, please see Chapter 27, *Subclassing*.

Chapter 9

Process Classes

9.1 Overview

A *process* class is defined as a class which encapsulates a business process. It is a collection of operations. Process classes do not have data maintenance operations, or indeed any relationship with database tables. Instead they manipulate other entity and process classes in order to implement a business process.

For example, in a banking system, you could have an account transfer process which debits money from one account and credits another. In this case, internally, the process would use the above Account entity class to update one account to debit it, followed by an update of the other account to credit it. Note that the process class itself does not do any database manipulation—it merely packages a sequence of entity operations in order to carry out the business process modelled.

Process classes have a stereotype of `<<process>>`.

9.2 Business Process Objects

Business Process Objects (*BPO*s) are the classes which reside in the *Business Object Layer* (*BOL*) of a Cúram server application, i.e. the architectural layer between the *Remote Interface Layer* (RIL) and the *Data Access Layer* (DAL). All business logic is implemented in this layer and as a result, BPOs constitute the large majority of handcrafted coding required to create a server application.

BPOs do not *directly* communicate with the RDBMS (implemented - largely automatically - in the DAL), nor the middleware (implemented - largely automatically - in the BOL): their job is specifically to implement business logic.

9.3 Rules

- Process classes must have a stereotype of <<process>>.
- Process classes cannot have aggregations to any other classes.
- Process classes can only inherit from other process classes - not from entity classes.
- Process classes cannot have attributes.
- Two or more process classes can have the same name provided that different CODE_PACKAGE values have been specified for each. See Section 3.2, *Options* for more details.

9.4 Operations

Within process classes there are four operations supported:

9.4.1 default

The <<default>> stereotype offers a standard or plain operation.

9.4.2 batch

For operations of stereotype <<batch>>, the Cúram generator will produce the necessary source code wrappers to build a batch wrapper program which will enable this operation to be run by the Batch Launcher (see the *Cúram Batch Processing Guide* for more details).

The rules when defining batch operations are:

- Batch operations cannot have more than one parameter.
- Parameters to batch operations must be structs.
- A process class cannot have more than one <<batch>> operation.

9.4.3 wmdpactivity

A method of a process class can be designated as a deferred processing method by setting its stereotype to << wmdpactivity>>.

For more information please consult the *Cúram Server Developer's Guide*.

9.4.4 qconnector

For operations of stereotype <<qconnector>>, the generator will produce the necessary source code to connect to a JMS provider (e.g.

). For more information please see Chapter 26, *Cúram JMS Queue Connectors*.

9.5 Options

9.5.1 Abstract

Specifies that the class is abstract. Abstract classes are intended to be subclassed by other classes. For more information on abstract classes and subclassing, please see Chapter 27, *Subclassing*.

9.5.2 Generate FIDs

Specifies that the class should have a Function Identifier generated for it.

9.5.3 Replace Superclass

This is only relevant if this process class has been subclassed from another class. For more information on subclassing, please see Chapter 27, *Subclassing*.

Chapter 10

Struct Classes

10.1 Overview

Struct classes are *Java* classes with public attributes and no modeled methods (It is the *Java* equivalent of a C++ struct.). They allow for the grouping of domain definitions and other struct classes to form programmatic record definitions.

Typically, struct classes are used as arguments to operations of entity and process classes. Structs are used to 'package' arguments in order to avoid long argument lists. Struct classes can also aggregate each other; these aggregations turn into struct members.

For example, in the case of a bank account entity, the parameters to a read operation would consist of a key struct and a details struct. The key struct might contain a single field for the account number. The details struct might have several fields including Name, Balance, etc.

Struct classes have a stereotype of `<<struct>>`.

10.2 Rules

- Struct classes must have one or more attribute or aggregation, i.e. a struct cannot be empty.
- Struct classes are not allowed to have operations.
- Struct attribute types must be defined in terms of valid domain definitions.
- Struct classes may aggregate entity classes or other struct classes.
- Struct classes are not allowed to be involved in inheritance relationships.
- Struct classes used as key or details parameters to non-standard database

operations must not aggregate other structs, i.e. they must be “flat”.

- Two or more struct classes can have the same name provided that different CODE_PACKAGE values have been specified for each, i.e. similarly named struct classes must be distinguishable by having CODE_PACKAGE settings. See Section 3.2, *Options* for more details.
- In most cases you will have to define a struct in order to use it as a parameter to an operation. The exception to this rule is standard key and details structs. These are generated automatically by the Cúram generator and are available for use by the developer.

10.3 Outputs

Input meta-model struct classes map directly onto generated *Java* classes in the <ProjectPackage>.<CodePackage>.struct package. The *Java* <<struct>> class contains public fields corresponding to each attribute defined in the model.

Each field is initialized to its default value - zero for numerics, empty string for Strings, etc. - so the developer does not have to worry about null values.

Each field is accompanied by comments describing the Domain Definition hierarchy for the datatype.

The class also contains generated code enabling the struct to be cloned and assigned to other structs.

Struct classes have no counterpart in generated DDL.

10.4 Options

10.4.1 Audit Fields

If this option is switched on then the available pre-configured audit fields will be automatically added to this struct.

This option should be enabled if the struct class is being used as a write operation of an entity which also has Audit Fields switched on.

For more information, see Chapter 4, *Audit Mappings Classes*.

Chapter 11

Attributes

11.1 Overview

Attributes represent fields of the underlying *Java* class. The class stereotype determines the attribute stereotypes that are valid for the class. The combination of class and attribute stereotypes will determine behavior such as how the Cúram generator processes the UML meta-model. The following sections provide details on how attributes work and need to be specified.

11.2 Attribute Rules

The following table shows the mapping of class stereotypes to attribute stereotypes.

Class Stereotype	Valid Attribute Stereotypes
audit_mappings	audit_mappings
domain_definition	N/A
entity	details, key
facade	N/A
listrdo	dataitem
loader	N/A
process	N/A
rdo	dataitem
struct	default
webservice	N/A
wsinbound	N/A

Table 11.1 Mapping of Class and Attribute Stereotypes

- Attribute names must be unique within a class.
- Attributes must be defined in terms of domain definitions.
- Since attributes ultimately appear in generated *Java* code, their names must be valid *Java* identifiers.
- The order of attributes in the primary key of an entity is determined by the order in which the attributes appear in the entity class. Since their order in the entity is not critical, you can change this order to obtain the primary key configuration you desire.

11.3 Attribute Options

11.3.1 Allow NULLs

This option is available only for `<<details>>` stereotyped attribute on an entity class.

It determines whether NULL values are permitted on the corresponding database field. Setting this option to `no` causes a `Not Null` qualifier to be included with this field in the generated DDL script.

The default value for this option is dependent on the underlying data type of the field. The default value of this option for the attributes for fields of type `SVR_BOOLEAN`, `SVR_CHAR`, `SVR_FLOAT`, `SVR_DOUBLE`, `SVR_MONEY`, `SVR_INT8`, `SVR_INT16`, `SVR_INT32` is `no`.

The default value of this option for the attributes of type `SVR_BLOB`, `SVR_DATE`, `SVR_DATETIME`, `SVR_STRING`, `SVR_INT64` is `yes`.

This topic is dealt with in Section 19.4.3, *Null Considerations*.

11.3.2 Multibyte Expansion Factor

This is an override for the domain-level `Multibyte Expansion Factor`, described in Section 5.2.11, *Multibyte Expansion Factor*, and is applicable to string entity attributes only.

It specifies an expansion factor (float from 1.0 to 4.0) to be applied when multi-byte character set (MBCS) data will be used with *DB2* or *DB2 for z/OS*. It operates in conjunction with its equivalent domain option and the global build-time properties `curam.db.multibyte.expansion.default.factor` and `curam.db.multibyte.expansion`, which are described in the *Cúram Server Developer's Guide*. This option is only necessary for *DB2* MBCS data in order to deviate from the global or domain settings. For example, you might choose to set this option to 1.0 (which effectively turns off expansion) for a string attribute where you know the contents will never contain localized data (e.g. they are constrained to programmatically defined Western characters and can't be input via a client). This option is ignored if

the feature is turned off via the `curam.db.multibyte.expansion` property.

Chapter 12

Operations

12.1 Overview

Operations represent the functionality of modeled classes, which, depending on their type, can be provided by the Cúram generator or "handcrafted".

12.2 Rules

- Operations must belong to either `<<entity>>`, `<<process>>`, `<<facade>>`, `<<webservice>>`, or `<<wsinbound>>` classes.
- Operations can be fully handcrafted or can make use of the facilities offered by standard operations. Standard operations are covered in detail in later sections.
- Operations cannot be individually hidden from or exposed to clients, only whole classes can be hidden or exposed.

These are the rules regarding the requirements for using structs (versus domain values) as parameters and return values for operations:

- Parameters for batch operations must be structs.
- Parameters and return types for all database operations must be structs.
- Parameters and return types for queue operations must be structs.
- Parameters and return types for web service connector operations must be structs.
- Parameters and return types for client-visible operations must be structs. (Domain parameters and return types are not supported by the HTML client.)
- Parameters and return types for other operation stereotypes including

web service client operations or other classes may be domain definitions.

12.3 Operation Options

12.3.1 Audit BI (Business Interface) Calls to this Operation

This is only relevant to client-visible operations.

This option specifies whether Business-Interface-level auditing should be performed for this operation. For Business-Interface-level auditing records the following information about the operation call is recorded:

- the operation name (Function Identifier);
- the username of the caller;
- the date and time;
- the transaction type (online/batch/deferred/etc.).

This option can be overridden at application startup time using application properties, this functionality is described with an example in the Database Table-level Auditing option description contained in this section.

12.3.2 Auto ID Field

This is only relevant for certain insert operations of entity classes.

Specify which field is to be used as the Auto ID field. The Auto ID field is automatically populated with a generated unique ID during the insert to ensure that the record can be uniquely identified.

12.3.3 Auto ID Key

This option is used only in conjunction with the `Auto ID Field` option.

It allows you to specify the key set from which a unique ID should be generated.

12.3.4 Business Date

This option is only relevant to operations of a process class.

It allows you to specify that one field of the operation parameters be treated as the Business Date Field for the operation. This means that the value of this parameter to the operation becomes the Business Date for the duration of the transaction. The Business Date is the `Date` or `DateTime` which gets returned by the following methods:

- `curam.util.transaction.TransactionInfo.getBusi-`

nessDateTime()

- curam.util.transaction.TransactionInfo.getBusinessDate()
- curam.util.type.Date.getCurrentDate()
- curam.util.type.DateTime.getCurrentDateTime()

The main purpose of this feature is to give greater flexibility when running batch programs for which processing dates are significant. Consider the example where a report generating program is run at the end of each day to count all payments issued that day. The payment records are obtained by reading all records whose issue date equals curam.util.transaction.TransactionInfo.getBusinessDate(). This program will process a different set of records depending on the day on which it is run.

Now consider what would happen if you needed to re-generate the report from 10 days ago.

Without the Business Date feature you would have to do the following:

- Submit a batch request for your batch program.
- Change the system date on the machine where the batch program will be run. Note that you will have to ensure that this doesn't affect anyone else, so nobody else can use the machine while the system date is being changed.
- Ensure that your batch request is the only one in the queue.
- Run the batch launcher to cause your batch program to be run.
- Revert the system date on the machine.
- Make the machine available for general usage again.

However if your batch program parameters include a Business Date field you need only do the following:

- Submit a batch request for your program, ensuring that the batch job parameter which has been specified as the Business Date is set to the date 10 days ago.
- Run the batch launcher.

Syntax for **Business Date** option:

The Business Date option should be specified in one of the following formats:

- fieldName
- *paramName*.fieldName

where

- *paramName* is the name of a parameter. This is optional and, if not specified, the first operation parameter is assumed.
- *fieldName* is the name of a field in the parameter struct.

Business Date example 1

- Struct ReportArguments contains a Date field named effectiveDate.
- The following is a batch operation: doReportGeneration(ReportArguments *arg1*).
- To use effectiveDate as the Business Date for the operation, you can set the Business Date option to either arg1.effectiveDate or because it is the first (and only) parameter: effectiveDate.

Business Date example 2

- Struct GeneratePaymentsParameters contains a Date field named paymentDate.
- The following is a batch operation: generatePayments(SomeStruct *argA*, GeneratePaymentsParameters *argB*).
- To use paymentDate as the Business Date for the operation, you would set the Business Date option to argB.paymentDate.

Rules for **Business Date** option:

- This option is only relevant to operations which correspond to individual server transactions. Such operations are the operations of facade classes and <<batch>> operations. Note that it is *not* applicable to workflow activity or deferred processing operations.
- The field which is specified as the Business Date Field must be of type SVR_DATE or SVR_DATETIME.
- The Business Date Field only takes effect when the operation is invoked by a remote client (either the HTTP client or a web services client) or by the *Batch Launcher*. It does not take effect for operations which are invoked directly from *Java* code. This is because the latter does not result in a new server transaction being started.
- If the Business Date Field is set to null, curam.util.type.Date.kZeroDate or curam.util.type.DateTime.kZeroDateTime for a method invocation, it is ignored and the Business Date does not get overridden for that transaction. In this case the Business Date for the transaction will be

either the current system date, or the overridden value specified in application properties - see the `Date` and `DateTime` JavaDoc documentation for more details.

12.3.5 BytesMessage encoding character set

This is only relevant for `<<qconnector>>` operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.6 Database Table-level Auditing

This is only relevant to database operations of entity classes.

This option specifies whether table-level auditing should be performed for this operation. Table-level auditing records detail information about the changes made to actual data on the database table.

The behavior of auditing depends on whether Optimistic Locking is switched on or off for the operation. For more information about Auditing, see Section 6.8, *Table Level Auditing*.

This option can be overridden at application startup time using application properties, this functionality is available to Audit BI Calls and this option and what follows is an example of how it should be used.

Changing operation auditing options without rebuilding.

Changes to operation options `Audit BI` and `Database Table-level auditing` in the model require a rebuild and redeploy to take effect. It is possible to override these properties in application properties whereby the changes take effect when the application is restarted.

These two options can be targeted at individual operations by specifying application properties whose format is as follows:

```
curam.audit.audittrail.<ProjectName>.<ClassName>.<OperationName>
```

```
curam.audit.opaudittrail.<ProjectName>.<ClassName>.<OperationName>
```

or, if the class is in a code package:

```
curam.audit.audittrail.<ProjectName>.<CodePackage>.<ClassName>.<OperationName>
```

```
curam.audit.opaudittrail.<ProjectName>.<CodePackage>.<ClassName>.<OperationName>
```

Properties whose names begin with `curam.audit.audittrail` apply to the Database Table-Level Auditing option and cause data to be captured to table AuditTrail.

Properties whose names begin with

`curam.audit.opauditttrail` apply to the Audit BI calls option and cause data to be captured to table OpAuditTrail.

Example (1): To switch on table level auditing for operation `modify` of entity `CaseHeader` which is in code package core of the Cúram application, set the property `curam.audit.audittrail.curam.core.CaseHeader.modify` to `true`.

Example (2):. To switch off operation auditing for operation `modifyAddress` of process class `Participant` which is in code package `core.facade` of the Cúram application, set the property `curam.audit.opauditttrail.curam.core.facade.Participant.modifyAddress` to `false`.

In Summary,

- changing the value of an auditing option requires an application restart to take effect
- The `curam.audit.opauditttrail.*` properties only affect client-visible operations.
- The `curam.audit.audittrail.*` properties only affect stereotyped entity operations - excluding stereotypes `<<ns>>` and `<<nsmulti>>`.

12.3.7 Field Level Security

Field Level Security can be applied for the fields returned by client-visible operations (i.e. operations of the Facade class). It is only relevant to operations of a client-visible operation (defined in a Facade class).

In *Rational Software Architect* the *Secure Fields* properties tab of the Facade class operation allows you to apply security to any field returned by an operation by specifying a security identifier (SID) for that field.

To establish secure returned fields for an an operation use the *Secure Fields* button from the properties tab for the operation. Clicking the *SID Name* cell for the returned *Field Name* allows you to enter the security identifier (SID). The maximum length of a security identifier is 100 characters.

The client infrastructure will then ensure that fields for which a SID has been specified can only be viewed by users to whom that SID has been granted. Fields for which no SID has been specified will be visible to all users.

All the information about Field Level Security - which SID is assigned to a field - is written by the generator to an XML file and is loaded into database table FieldLevelSecurity by the Data Manager. The Data Manager configuration file `datamanager_config.xml` must be changed to reference the generated file `<ProjectName>_FieldsReturned.xml`. This can be done by adding an entry to the **initial** target as shown in Example 12.1, *Sample datamanager_config.xml for adding field level security information*

to the database below.

```
<target name "initial"
<entry
  name="build/svr/gen/ddl/<ProjectName>_Fids.xml"
  type="xml" base="basedir" />
<entry
  name="build/svr/gen/ddl/<ProjectName>_FieldsReturned.xml"
  type="xml" base="basedir" />
</target>
```

Example 12.1 Sample datamanager_config.xml for adding field level security information to the database

Once the field names and SIDs have been added to the FieldLevelSecurity table, the SIDs should be loaded into the SecurityIdentifier to enable them to be assigned to groups. This can be done using the database command shown in Example 12.2, *Inserting field level security SIDs into the infrastructure SecurityIdentifier table* below.

```
INSERT INTO SecurityIdentifier(sidName, sidType, versionNo)
  SELECT DISTINCT sidName, 'FIELD', 1 from FieldLevelSecurity
  WHERE sidName IS NOT NULL;
```

Example 12.2 Inserting field level security SIDs into the infrastructure SecurityIdentifier table

These SIDs can then be assigned to user groups using the Security Administration console.

12.3.8 JNDI name of the QueueConnectionFactory class

This is only relevant for <<qconnector>> operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.9 JNDI name of the transmission queue

This is only relevant for <<qconnector>> operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.10 JNDI name of the reply queue

This is only relevant for <<qconnector>> operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.11 Message type

This is only relevant for <<qconnector>> operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.12 No Generated SQL

This is only relevant to database operations of entity class.

Switches off generation of data access code, allowing developers to provide their own implementation.

For example, if `No Generated SQL` was set to `yes` for a standard read operation named `myRead` the generator will produce a declaration of an abstract method named `myRead_da` with the same signature as the formerly generated `myRead` method. The developer must provide the implementation of method `myRead_da` as is shown in the following listing:

```
public MyEntityDtls myRead_da(
  final MyEntityKey key, final boolean forUpdate)
  throws AppException, InformationalException {

  final MyEntityDtls result = new MyEntityDtls();
  result.idNumber = "1234";
  return result;
}
```

Example 12.3 Handcrafted data access implementation for a standard read

For readmulti operations - i.e. operations of stereotype `<<readmulti>>`, `<<nsreadmulti>>`, `<<nkreadmulti>>` or `<<nsmulti>>` - the handcrafted implementation must follow a different pattern. The method is declared as returning a list struct *but this return value is ignored*. Readmulti operations in Curam are implemented using the visitor design pattern whereby a subclass of `curam.util.dataaccess.ReadmultiOperation` is passed into the data access operation which then invokes its `operation(Object)` for each record found. Usually this operation will add the record to a collection which gets returned to the caller. This is described in greater detail in the *Cúram Server Developers Guide*.

The key point is that for readmulti operations, data is returned to the caller by adding it to the `ReadmultiOperation` class by calling its `operation(Object)` method, and *not* by simply returning it from the method. This is shown in the following example:

```
/*
 * This implementation returns two hard coded dummy records.
 */
public MyEntityDtlsList readmulti_da(
  final SomeKey k, final ReadmultiOperation op,
  final boolean requireInformational)
  throws AppException, InformationalException {

  // Create and add one record for return to the caller.
  final MyEntityDtls oneDtls = new MyEntityDtls();
  oneDtls.idNumber = "2222";
  op.operation(oneDtls);

  // Create and add another record for return to the caller.
  final MyEntityDtls twoDtls = new MyEntityDtls();
  twoDtls.idNumber = "3333";
  op.operation(twoDtls);

  // our return value is ignored so just return null.
  return null;
```

```
  }
```

Example 12.4 Handcrafted data access implementation for a readmulti

12.3.13 On Fail Operation

This is only relevant for database operations of entity classes.

This option switches on the on-fail exit point.

If any error occurs in the Data Access Layer (DAL), this function is invoked with a copy of the parameters given to the DAL and a copy of the DAL exception corresponding to the error.

The type of exception depends on the type of error which occurred. The error can either be handled in this exit point or the exception can be thrown from here to allow the error to be handled elsewhere.

For more information on Exit Points, see Section 6.9, *Exit Points*.

12.3.14 Optimistic Locking

This is only relevant for certain update operations of entity classes.

This option switches on optimistic locking for this operation.

Note that this option is only allowable if the `Allow Optimistic Locking` option has been set for the entity class.

For more information on optimistic locking, see Section 6.7, *Concurrency Control - Optimistic Locking*.

12.3.15 Order By

This option applies only to entity operations of stereotype `<<readmulti>>`, `<<nsreadmulti>>` and `<<nkreadmulti>>`.

This option allows you to specify the fields by which a sequence of records are sorted as they are read from the database. Any or all of the fields of an entity are valid arguments for this option. Records are always sorted in ascending order.

If this option is not specified, records will be returned in arbitrary order.

12.3.16 Post Data Access Operation

This is only relevant for database operations of entity classes.

This option determines whether a standard database operation has a post-exit point.

For more information on exit points, see Section 6.9, *Exit Points*.

12.3.17 Pre Data Access Operation

This is only relevant for database operations of entity classes and determines whether a database operation has a pre-exit point.

For more information on exit points, see Section 6.9, *Exit Points*.

12.3.18 Readmulti_Max

This option applies only to entity operations of stereotype `<<readmulti>>`, `<<nsreadmulti>>`, `<<nkreadmulti>>` and `<<nsmulti>>`.

This allows you to specify the maximum number of records returned by a readmulti operation. If there are more records available than the Readmulti_Max, then handling is based on the setting of the Treat Readmulti_Informational option (Section 12.3.19, *Readmulti_Informational*).

If this option is not specified then the generator system default will be used.

Specifying a value of `0` for this option is interpreted as `infinity` and no limit will be applied to the number of records returned.

12.3.19 Readmulti_Informational

This option applies only to entity operations of stereotype `<<readmulti>>`, `<<nsreadmulti>>`, `<<nkreadmulti>>` and `<<nsmulti>>`.

This allows you to determine the handling of the system when the specified Readmulti_Max is reached (Section 12.3.18, *Readmulti_Max*). The default behavior is that a Readmulti_Max message gets logged and all entries are returned to the user. If this option is specified, then an InformationalMessage can be added to the current transactions InformationalManager, for handling in the Application's Facade layer. In this case only the specified Readmulti_Max number of entries will be returned to the user.

If this option is not specified then the generator system default `false` will be used.

12.3.20 Response message timeout (seconds)

This is only relevant for `<<qconnector>>` operations of process classes. See Section 26.3, *Options on qconnector Operations* for more information.

12.3.21 Security

This is only relevant to client-visible records.

This option determines whether security will be applied to this operation. If security is switched on for an operation, then the generator will generate

code in the RIL which checks whether the user is authorized to invoke the operation. If the user is not authorized to invoke the operation an exception will be thrown.

12.3.22 SQL

This is only relevant to entity operations of stereotype `<<ns>>` and `<<nsmulti>>`.

This option allows the developer to supply the SQL code to be executed by the operation. The generator converts the supplied SQL into DAL (*Java* and SQL) code.

For more information about this option, see Section 19.4, *Using Handcrafted SQL in Non-Standard Entity Operations*.

12.3.23 Transactional

This allows you to specify whether a transaction is started for an operation. This is only relevant to client-visible operations.

12.3.24 Where

This is only relevant to `<<readmulti>>` and `<<nsreadmulti>>`operations of entity classes.

This allows the developer to specify a custom `WHERE` clause for the generated SQL used by the DAL code for this operation.

12.4 Operation Parameter Options

12.4.1 Mandatory Fields

This option allows the developer to specify mandatory fields for any given parameter. Mandatory fields are fields that must be populated when displayed on a client page.

The option value must be populated with a single line, comma delimited string.

Consider the following operation:

```
public interface Employer
{
  public void updateEmployerDetails(
      PersonDetails personDtls
      EmploymentDetails employmentDetails)
    throws AppException, InformationalException;
}
```

Example 12.5 Operation Signature

The pseudo-code for the structures involved as parameters in this operation is outlined below:

```
// Note that since a person can have two addresses,
// PersonDetails aggregates AddressDetails twice
// - "homeAddress" and "workAddress".
struct PersonDetails {
  String firstName;
  String surname;
  AddressDetails homeAddress;
  AddressDetails workAddress;
}
// The role name for the struct aggregation between
// PersonDetails and AddressDetails "homeAddress" struct
// is set to "homedtls"
struct AddressDetails {
  String addressLine1;
  String addressLine2;
  String city;
  String country;
}
// Note that EmploymentDetails aggregates AddressDetails once.
// The role name for the struct aggregation between
// EmploymentDetails and AddressDetails "employerAddress"
// struct is set to "employmentdtls"
struct EmploymentDetails {
  String employerName;
  Date employmentStartDate;
  AddressDetails employerAddress;
}
```

Example 12.6 Pseudo-Code for Parameter Structures

In this example, we want to make the following fields mandatory for our operation parameter:

For the `personDtls` parameter:

- the person's first name; and
- the first line of the `PersonDetails` home address.

For the `employmentDetails` parameter:

- the first line of the employer's address.

Set the `Mandatory Fields` option of parameter `personDtls` to:

`firstName`, `homeAddress.addressLine1`

Set the `Mandatory Fields` option of parameter `employmentDetails` to:

`employerAddress.addressLine1`

Therefore, if adding mandatory fields that are contained in structures aggregated by the parameter type class, they must be fully qualified by the relevant aggregation role names as shown above.

Chapter 13

Entity Operations Overview

13.1 Introduction

This chapter provides an overview of entity operations, which are covered in more detail in the following sections and chapters.

In the SDEJ, a database operation is an operation of an entity class whose stereotype is recognized by the Cúram generator. For these operations the generator will generate data-access *Java* code based on the stereotype.

The generator treats all other operations as if their stereotype was blank and will produce *Java* interfaces and factories for them, but does not generate any data-access code for these operations. The developer must then provide the implementation.

13.2 Standard Operations

13.2.1 Standard Single-Record Operations

Standard single-record operations are the most basic type of operation provided in that only a single row from the database is returned and no arguments are required to be modelled as the code generator assumes standard key and details structs where appropriate.

These operations are represented by the following operation stereotypes:

- `<<insert>>`
- `<<modify>>`
- `<<read>>`
- `<<remove>>`

13.2.2 Standard Multi-Record Operations

Rather than operating on a single database table row, these operations allow for processing multiple rows. In database maintenance applications, it is often necessary to return multiple records to a user interface, from which the user selects one for processing. Batch programs also frequently operate on multiple rows of a table; for example a Bank account statement printing batch program will typically operate on the accounts of every client on record.

These operations are represented by the following operation stereotype:

- `<<readmulti>>`

13.3 Non-Standard Operations

13.3.1 Generated SQL Operations

Non-standard generated SQL operations are similar to the standard operations except that the arguments and return type are not assumed to be standard key and standard details structs. The developer is required to specify a struct for each argument and return type.

The attributes of the argument and return structs must be subsets of the fields of the entity.

The argument structs can be user-defined structs from the input meta-model, or the generated standard structs that are not explicitly defined in the input meta-model. Using generated standard key and details structs as the parameters to non-standard operations is equivalent to simply using standard operations.

It is important to remember that since the key struct of a non-standard generated SQL operation is defined by the developer, it is possible to define a key struct which does not uniquely identify a single record. If this happens, certain operations may not behave as expected. For example, in the case of a non-standard modify operation, all records matching the key will be modified, not just the intended record.

These operations are represented by the following operation stereotypes:

- `<<nsinsert>>`
- `<<nsmodify>>`
- `<<nsread>>`
- `<<nsreadmulti>>`
- `<<nsremove>>`

13.3.2 Handcrafted SQL Operations

Non-standard handcrafted SQL operations are the most flexible type of operation provided by the generator. They allow the developer to specify custom parameters and SQL for the operation. No parameters are generated for <<ns>> operations except those provided by the developer. All parameters provided by the developer are replicated into all the generated layers of the application.

This type of operation is intended to be used for situations where none of the other operations are suitable. This includes joins across tables and queries which count or calculate max, etc.

These operations are represented by the following operation stereotypes:

- <<ns>>
- <<nsmulti>>

13.4 Non-Key Operations

Non-key operations operate on all rows of a database table and so would typically be used on tables containing one row.

These operations are represented by the following operation stereotypes:

- <<nkmodify>>
- <<nkread>>
- <<nkreadmulti>>
- <<nkremove>>

13.5 Batch Operations

Batch programs, as described in the *Cúram Batch Processing Guide* have operations available to them that are tailored specifically to the batch environment.

These operations are represented by the following operation stereotypes:

- <<batchinsert>>
- <<batchmodify>>

Chapter 14

Entity Insert Operations

14.1 Overview

Insert operations as their name suggests insert, or add, a row onto a database table. Therefore, by definition, they operate on only a single row at a time. There are two types:

- `<<insert>>`
- `<<nsinsert>>`

14.2 Standard Insert

A standard insert operation has a stereotype `<<insert>>`.

14.2.1 Description

Standard insert operations insert a single record onto the appropriate database table using the information passed in a standard details struct. No arguments are required to be specified for these operations in the input model. Extra arguments can be specified and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

14.2.2 Use

You should use a standard *insert* operation when you want to create a new record on a database table, and you want to update each attribute.

No arguments are required to be specified for this operation in the input model. Generated standard key and details structs are assumed as arguments where appropriate.

Extra arguments can be specified for this operation and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

This pattern can also be used in conjunction with the Auto ID Field sequence number generation pattern.

14.2.3 Parameter and Generator Notes

Standard insert operations use the entity's details structure as an input parameter—this is automatically generated and contains all the fields of the record.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will add the standard details struct as a parameter.

14.3 Non-standard Insert (Generated SQL)

A non-standard insert operation has a stereotype `<<nsinsert>>`.

14.3.1 Description

Non-standard insert operations insert a single record onto the database table of the parent entity with information from a non-standard details struct provided by the developer.

14.3.2 Use

You should use a *non-standard insert* operation when you want to create a new record on a database table, and you do not need to update each attribute. Attributes not specified in the parameter to a non-standard insert are set to null values on the database.

Non-standard insert operations are more efficient than standard inserts, because there is less I/O to the database. It is the responsibility of the application designer to decide whether the improved efficiency is worth the extra complexity of having more operations on your entities.

You might choose to use a non-standard insert where you know the database can perform the operation significantly more efficiently or where the operation will be used by a very high volume transaction.

Non-standard insert operations take a single input parameter - a structure defining the attributes to be inserted. Each attribute of this structure must match some entity attribute by name and type.

14.3.3 Parameter and Generator Notes

A Warning is displayed if a non-standard operation has non-standard details parameter, which does not include fields that cannot be null. Refer to Section 19.4.3, *Null Considerations*.

Fields which are not included in the details struct will not be initialized, i.e. they will be set to <null> by the DBMS.

- *Parameters* - A non-standard details struct.
- *Return Value* - None.
- *Generator action* - None.

Chapter 15

Entity Read Operations

15.1 Overview

Read operations obtain one or more (multi) rows from the database table, depending on the type of operation and arguments provided. These are the read operation types:

- `<<read>>`
- `<<readmulti>>`
- `<<nsread>>`
- `<<nsreadmulti>>`
- `<<nkread>>`
- `<<nkreadmulti>>`

15.2 Standard Read

A standard read operation has a stereotype of `<<read>>`.

15.2.1 Description

Standard read operations read a single record from a database table into a standard details struct, using a standard generated key struct (i.e., the primary key) as search criteria. No arguments are required to be specified for these operations in the input model. Extra arguments can be specified and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

15.2.2 Use

You should use a standard singleton *read* operation when you want to read all of the attributes of a specific database record. Standard singleton read operations use the primary key of an entity to locate the target record. You cannot create standard singleton read operations for entities that do not have primary keys. Since the primary key of an entity is unique, a standard singleton read always returns a single database record.

15.2.3 Parameter and Generator Notes

Standard singleton read operations use the entity's key and details structures as input and output parameters respectively - these are automatically generated and are not specified in the UML meta-model.

No arguments are required to be specified for these operations in the input model. Generated standard key and details structs are assumed as arguments where appropriate.

Extra arguments can be specified and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will add the standard key struct as a parameter and the standard details struct as the return value.

15.3 Standard Readmulti

A standard readmulti operation has a stereotype `<<readmulti>>`.

15.3.1 Description

Standard multiple read operations use an input parameter which you designate as the key structure for the operation. The return value is a structure containing a list of the entity's details structures. You must specify the first parameter, but since the return value is automatically generated, it is not specified in the UML meta-model.

15.3.2 Use

You should use a standard *multiple read* operation when you want to read all of the attributes of a set of database records, based on a key that you specify. The stipulation about efficiency of keyed access, as described for non-standard read, modify and remove operations, applies equally well to multiple reads - it is up to the designer to ensure efficient use of database indices.

15.3.3 Parameter and Generator Notes

A standard readmulti operation takes a partial key struct, and returns a list of standard details structs; every record matching the criteria is returned in the list.

By default, the records in a readmulti are unsorted and are returned in arbitrary order.

This can be changed by using the `Order By` option of the readmulti operation. This option takes a list of the fields of the entity and sorts them in ascending order.

- *Parameters* - A key struct to specify the search criteria for which record(s) to retrieve. The members of the struct must be a subset of the standard details struct for the entity.
- *Return value* - None.
- *Generator action* - The generator will create a list wrapper for the standard details struct for the entity, and add this as the return value for the operation.

15.4 Non-standard Read (Generated SQL)

A non-standard read operation has a stereotype `<<nsread>>`.

15.4.1 Description

Non-standard read operations read a single record from a database table into a details struct, using an key struct as search criteria.

15.4.2 Use

You should use a *non-standard read* operation when you want to read a subset of the attributes on a database record, or you want to use a key other than the primary key of the entity. Non-standard operations use a key that you specify to locate the target record. It is not possible to guarantee at development time that only one record will be targeted. If there is more than one record in the result set, a runtime errors is generated.

Non-standard read operations can be more efficient than standard ones because they result in less database I/O.

As with any operation where you specify the key yourself, there is no guarantee that the database will be able to access the target records efficiently - it is up to the designer to ensure that appropriate indices are defined to ensure this.

15.4.3 Parameter and Generator Notes

Non-standard read operations use key and details structures - as input and return types respectively - that you must create yourself and specify as operation parameters in the UML meta-model. Each attribute of each of these structures must match some entity attribute by name and type. It is possible to use standard (generated) key or details structures also.

- *Parameters* - A non-standard key struct to specify the record to be retrieved. The key must be capable of uniquely identifying a single record. If more than one record matches the criteria, an exception will be thrown.
- *Return Value* - A non-standard details struct into which the data is retrieved.
- *Generator action* - None.

15.5 Non-standard Readmulti (Generated SQL)

A non-standard readmulti operation has a stereotype `<<nsreadmulti>>`.

15.5.1 Description

Non-standard readmulti operations take a partial key struct and a details struct as input meta-model parameters. They return a list of the provided details struct; every record matching the criteria is returned in the list.

The only difference between a non-standard readmulti and a standard readmulti is that a non-standard readmulti must specify a return value whereas for a standard readmulti this is assumed to be the standard generated details struct for the entity. For non-standard readmulti the developer is required to specify a struct as the return value of the operation. The fields of this struct must be a subset of the fields of the entity.

15.5.2 Use

You should use a *non-standard multiple read* operation when you want to read a subset of the attributes of a set of database records, based on a key that you specify. It is up to the designer to ensure efficient use of database indices when reading based on this key.

15.5.3 Parameter and Generator Notes

Like standard operations, non-standard multiple read operations use an input parameter which you designate as the key structure for the operation. The return value that you specify in the UML meta-model is a structure containing the attributes that you want returned for each record read from the database. The return value in the generated code is a list of the structure that you specified in the meta-model (the structure containing the list is automatically generated).

- *Parameters* - A non-standard key struct to specify the search criteria for which record(s) to retrieve. The members of the struct must be a subset of the fields of the entity.
- *Return Value* - A non-standard details struct to specify which attributes are to be returned from the readmulti operation. The members of this struct must be a subset of the fields of the entity.
- *Generator action* - The generator will create a list wrapper for the non-standard details struct specified by the developer, and use this as the return value for the operation.

15.6 Non-key Read

A non-key read operation has a stereotype `<<nkread>>`.

15.6.1 Description

Non-key read operations read the only record from a database table into a standard details struct.

Non-key operations, as the name suggests, do not take a key parameter. They operate by executing SQL statements which do not have a `where` clause; i.e. they operate on all rows on a table.

For a non-key read operation, there should be a single row on the table - this type of operation is typically used to read a value from a control table which contains a single record.

There is no such thing as a non-key insert operation since insert operations do not require a key parameter.

15.6.2 Use

You should use a *non-key singleton read* operation when you want to read a record from a database table on which there is a single record. A runtime error is generated if the database table contains more than one record.

This operation type is typically used for control tables containing a single record.

15.6.3 Parameter and Generator Notes

Non-key singleton read operations take no parameters and the generator automatically adds the standard details struct for the entity as the return type.

If more than one record exists on the table, an exception is thrown.

- *Parameters* - None.
- *Return value* - None.

- *Generator action* - The generator will add the standard details struct as the return value.

15.7 Non-key Readmulti

A non-key readmulti operation has a stereotype `<<nkreadmulti>>`.

15.7.1 Description

Non-key readmultis are very similar to standard readmulti operations, the only difference being that they return all rows of a table rather than those which match a partial key. They operate by executing SQL statements which do not have a `where` clause; i.e. they operate on all rows on a table.

Non-key operations, as the name suggests, do not take a key parameter. They operate by executing SQL statements which do not have a `where` clause; i.e. they operate on all rows on a table.

For a non-key read operation, there should be a single row on the table - this type of operation is typically used to read a value from a control table which contains a single record.

There is no such thing as a non-key insert operation since insert operations do not require a key parameter.

15.7.2 Use

You should use a *non-key multiple read* operation when you want to read all of the attributes of all of the records on a database table.

15.7.3 Parameter and Generator Notes

Non-key multiple read operations take no key argument. The return value is a structure containing a list of the entity's details structures as an output parameters. You specify no parameters in the UML meta-model (effectively the same interface and behavior as a standard multiple read except there is no key argument).

Generated non-key readmulti operations in the RIL and BOL have one parameter; a list details struct, i.e. a list of standard details structs for the entity.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will create a list wrapper for the standard details struct for this entity, and use this as the return value.

Chapter 16

Entity Update Operations

16.1 Overview

Update operations modify data in one or more rows of the database table, depending on the type of operation and arguments provided. These are the update operation types:

- `<<modify>>`
- `<<nsmodify>>`
- `<<nkmodify>>`

16.2 Standard Modify

A standard modify operation has a stereotype `<<modify>>`.

16.2.1 Description

Standard modify operations modify a specific record on an database table. No arguments are required to be specified for these operations in the input model. The record to be modified is specified using a generated standard key struct and the modified data is contained in a generated standard details struct. Extra arguments can be specified and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

16.2.2 Use

You should use a standard *modify* operation when you want to update all the attributes on a specific database record. Standard modify operations use the primary key of an entity to locate the target record. You cannot create stand-

ard modify operations for entities that do not have primary keys. Since the primary key of an entity is unique, a standard modify always updates a single database record.

The standard modify pattern can also be used in conjunction with the Optimistic Locking pattern.

16.2.3 Parameter and Generator Notes

Standard modify operations use the entity's key and details structures as input parameters - these are automatically generated and are not specified in the UML meta-model.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will add the standard key struct and standard details struct as parameters.

16.3 Non-standard Modify (Generated SQL)

A non-standard modify operation has a stereotype `<<nsmodify>>`.

16.3.1 Description

Non-standard modify operations update records on the database table of the parent entity with information from a non-standard details struct provided by the developer.

16.3.2 Use

You should use a *non-standard modify* operation when you want to update a subset of the attributes on a database record or records. Non-standard modify operations use a key that you specify to locate the target records, and this may result in multiple records being updated. You also specify which attributes of the entity are to be updated.

Non-standard modify operations can be more efficient than standard ones because they result in less database I/O, and the database may not have to update as many indices as would be the case for a standard modify operation.

16.3.3 Parameter and Generator Notes

Non-standard modify operations use non-standard key and details structures as input parameters that you must create yourself and specify as operation parameters in the UML meta-model. Each attribute of each of these structures must match some entity attribute by name and type. It is possible to use standard (generated) key or details structures also.

- *Parameters* - A non-standard key struct to specify the record to be modified. Note that this non-standard key may specify multiple records. In this case, all records matching the non-standard key will be updated.

  A non-standard details struct containing the updated version of the data.
- *Return Value* - None.
- *Generator action* - None.

16.4 Non-key Modify

A non-key modify operation has a stereotype `<<nkmodify>>`.

16.4.1 Description

Non-key modify operations modify all records on a database table with the information from a standard generated details struct.

Non-key operations, as the name suggests, do not take a key parameter. They operate by executing SQL statements which do not have a `where` clause; i.e. they operate on all rows on a table.

For a non-key read operation, there should be a single row on the table - this type of operation is typically used to read a value from a control table which contains a single record.

There is no such thing as a non-key insert operation since insert operations do not require a key parameter.

16.4.2 Use

You should use a *non-key modify* operation when you want to update all of the records on a database table. The attribute values of each record are set to those you specify in the parameter to the non-key modify function.

Typically you would only use a non-key modify operation for control tables containing only one record.

16.4.3 Parameter and Generator Notes

Non-key modify operations use the entity's details structure as an input parameter - this is automatically generated and is not specified in the UML meta-model.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will add the standard details struct as a parameter.

Chapter 17

Entity Delete Operations

17.1 Overview

Delete operations remove one or more rows from the database table, depending on the type of operation and arguments provided. These are the delete operation types:

- `<<remove>>`
- `<<nsremove>>`
- `<<nkremove>>`

17.2 Standard Remove

A standard remove operation has a stereotype `<<remove>>`.

17.2.1 Description

Standard remove operations delete a specific record from a database table. No arguments are required to be specified for these operations in the input model. The record to be deleted is specified using a generated standard key struct. Extra arguments can be specified and these arguments can be accessed by exit points for the operation, they do not have any effect on any of the generated code.

17.2.2 Use

You should use a standard *remove* operation when you want to delete a specific database record. Standard remove operations use the primary key of an entity to locate the target record. You cannot create standard remove operations for entities that do not have primary keys. Since the primary key of an

entity is unique, a standard remove always deletes a single database record.

17.2.3 Parameter and Generator Notes

Standard remove operations use the entity's key structure as an input parameter - this is automatically generated and is not specified in the UML meta-model.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - The generator will add the standard key struct as a parameter.

17.3 Non-standard Remove (Generated SQL)

A non-standard remove operation has a stereotype `<<nsremove>>`.

17.3.1 Description

Non-standard remove operations delete records from the database table of the parent entity matching the information in a key struct provided by the developer.

17.3.2 Use

You should use a *non-standard remove* operation when you want to delete a database record or records, based on a key that you specify. If the key you specify is not unique, multiple database records are deleted.

As with any operation where you specify the key yourself, there is no guarantee that the database will be able to access the target records efficiently - it is up to the designer to ensure that appropriate indices are defined to ensure this.

17.3.3 Parameter and Generator Notes

Non-standard remove operations use a key structure as an input parameter that you must specify in the UML meta-model. Each attribute of this key must match some entity attribute by name and type.

Note

When using segmented tablespaces with *DB2 for z/OS* (which are the default for version 9), IBM has changed the behavior of the JDBC driver as per this Technote: http://www-01.ibm.com/support/docview.wss?uid=swg21244002 Therefore, a `RecordNotFoundException` error will not be thrown when a negative row count is returned (i.e., a `DELETE`

FROM with no predicate).

- *Parameters* - A non-standard key struct to specify the record to be modified. Note that this non-standard key may specify multiple records. In this case, all records matching the non-standard key will be deleted.
- *Return Value* - None.
- *Generator action* - None.

17.4 Non-key Remove

A non-key remove operation has a stereotype <<nkremove>>.

17.4.1 Description

Non-key remove operations remove all the records from a database table.

Non-key operations, as the name suggests, do not take a key parameter. They operate by executing SQL statements which do not have a where clause; i.e. they operate on all rows on a table.

For a non-key read operation, there should be a single row on the table - this type of operation is typically used to read a value from a control table which contains a single record.

There is no such thing as a non-key insert operation since insert operations do not require a key parameter.

17.4.2 Use

You should use a *non-key remove* operation when you want to delete all of the records from a database table.

17.4.3 Parameter and Generator Notes

Non-key remove operations take no parameters.

- *Parameters* - None.
- *Return value* - None.
- *Generator action* - None.

Chapter 18

Entity Batch Operations

18.1 Overview

Batch operations are for inserting or removing a large number of records, typically for performance reasons. More information on *IBM Cúram Social Program Management* batch processing can be found in the *Cúram Batch Processing Guide*. These are the batch operation types:

- `<<batchinsert>>`
- `<<batchmodify>>`

18.2 BatchInsert

A batch insert operation has a stereotype `<<batchinsert>>`.

18.2.1 Description

Batch insert operations are intended to be used whenever a large amount of records are to be inserted into the database. By batching operations together, the number of round trips to the database is reduced and performance is improved.

Batch insert operations have a similar signature to non-standard insert operations and can be called in the same way. However, when a batch insert is invoked the record is not written immediately to the database. The insert statement is instead added to a batch of statements stored locally by the Cúram infrastructure by calling the `java.sql.PreparedStatement.addBatch` method. Once the batch has reached the desired size, it must be executed by calling the `$execute` method of the operation.

Note

The `$execute` method is never called automatically. It must be called from code written by the developer. If the entity object is destroyed without calling its `$execute` method, any pending (not executed) batched inserts will be discarded.

This means that batched inserts or modifies cannot be spread across multiple client invocations in an online environment because all entity objects are destroyed at the end of each invocation (transaction).

The `$execute` method of the operation calls the `executeBatch` method of `java.sql.PreparedStatement` and returns the result of this call which is an array of integers (int []). Each entry in this array corresponds to one statement in the batch and indicates how many records were affected by that statement. For example, for a successful batch of inserts, each entry of the array should be `1` to indicate that each statement caused one record to be written to the database. If one statement violated a unique constraint, its corresponding array entry would contain a zero. A returned value of `java.sql.Statement.EXECUTE_FAILED` indicates that the command failed to execute successfully.

The JDK documentation for `java.sql.PreparedStatement` provides further details regarding the information in this array, and how queued statements are executed.

The maximum number of statements in a batch is determined by the application property `curam.db.batch.limit` (default value = 30), or can be set for an individual operation by calling its `$setBatchSize(int)` method. The optimal size of a batch depends on many factors such as record size, database configuration and database vendor and can be different for each individual batch operation. It is the responsibility of the developer or DBA to determine this value.

If the batch limit is exceeded, an `AppException (INFRASTRUCTURE.ID_BATCH_SIZE_LIMIT_HAS_BEEN_REACHED)` is thrown by the batch insert operation. In this case the developer should simply call the `$execute` method of the operation, and then continue as before.

18.2.2 Use

You should use a `<<batchinsert>>` operation when you wish to insert a large number of records to the same entity in a single transaction.

18.2.3 Parameter and Generator Notes

A `<<batchinsert>>` operation takes a single input parameter - a structure defining the attributes to be inserted. Each attribute of this structure must match some entity attribute by name and type.

A warning is displayed if a `<<batchinsert>>` operation has non-standard details parameter, which does not include fields that cannot be null. Refer to section in Section 19.4.3, *Null Considerations*.

- *Parameters* - A non-standard details struct.
- *Return Value* - None.

Generator Action - The generator adds the following methods to a class containing a batch insert operation:

- `public void operationName$setBatchSize(final int newBatchLimit)`- This method sets the batch limit for the operation (overrides the value of the `curam.db.batch.limit` property).
- `public int[] operationName$execute() throws AppException, InformationalException` - This method executes the currently queued batch of statements for the operation.

18.3 BatchModify

A batch modify operation has a stereotype `<<batchmodify>>`.

18.3.1 Description

Batch modify operations are similar to batch insert operations except, as the name suggests, they are used to modify existing records rather than to insert new records.

18.3.2 Use

You should use a *batch modify* operation when you wish to modify a large number of records on the same entity in a single transaction.

18.3.3 Parameter and Generator Notes

A *batch modify* operation uses non-standard key and details structures as input parameters that you must create yourself and specify as operation parameters in the UML meta-model. Each attribute of each of these structures must match some entity attribute by name and type. It is possible to use standard (generated) key or details structures also.

A warning is displayed if a `<<batchmodify>>` operation has non-standard details parameter, which does not include fields that cannot be null. Refer to section in Section 19.4.3, *Null Considerations*.

- *Parameters* - A non-standard key struct to specify the record to be modified. Note that this non-standard key may specify multiple records. In this case, all records matching the non-standard key will be updated.

  A non-standard details struct containing the updated version of the data.
- *Return Value* - None.

Generator action - The generator adds the following methods to a class con-

taining a batch modify operation:

- `public void operationName$setBatchSize(final int newBatchLimit)`- This method sets the batch limit for the operation (overrides the value of the `curam.db.batch.limit` property).
- `public int[] operationName$execute() throws AppException, InformationalException` - This method executes the currently queued batch of statements for the operation.

Note

`<<batchmodify>>` operations cannot be spread across multiple client-server invocations (transaction). A `<<batchmodify>>` operation can only be used in an online transaction if the batch is executed before the end of the transaction.

Chapter 19

Entity Handcrafted SQL Operations

19.1 Overview

Using non-standard ("ns") operations "handcrafted" SQL can be utilized against the database. These are the non-standard operation types:

- `<<ns>>`
- `<<nsmulti>>`

19.2 Non-standard

A non-standard operation has a stereotype `<<ns>>`.

19.2.1 Description

All parameters for a `<<ns>>` operation must be structs. This is because the parameters are replicated in the Data Access Layer (DAL) and the DAL allows parameters to be structs only.

The return value for a `<<ns>>` operation must also be a struct. Similar to parameters for `<<ns>>` operations, the DAL allows return values to be structs only.

The developer *must* provide SQL with all `<<ns>>` operations; no SQL is automatically generated.

Non-standard operations must belong to an entity class. However, the SQL query can operate on any database table, it does not have to operate on only the database table belonging to the entity class; i.e., it can be used to perform SQL joins across tables.

For details on how to specify SQL in an operation, see Section 19.4, *Using Handcrafted SQL in Non-Standard Entity Operations*.

19.2.2 Use

You should use a *non-standard* operation for a database operation which is too complex for any of the above operations and which does *not* retrieve multiple records. Examples of such operations are:

- queries whose `where` clause contains comparisons other than equals, such as less-than, greater-than, etc.;
- queries or commands which operate on more than one database table;
- queries which return something other than an attribute of a table, such as the results of `max` and `count` functions.

The developer must specify the SQL to be executed and can specify zero or many parameters for the operation. All parameters must be structs and must be flat, i.e. they cannot aggregate other structs.

The handcrafted SQL can perform any database operation provided that a cursor is not required. This includes single-record-reads, single or multiple record updates and deletes, and joins across multiple database tables. This is because the parameter structs cannot aggregate other structs. If your hand-crafted SQL requires a cursor then an `<<nsmulti>>` operation should be used.

19.2.3 Parameter and Generator Notes

- *Parameters* - Struct(s).
- *Return value* - Struct.
- *Generator action* - None.

19.3 Non-standard multi

A non-standard multi operation has a stereotype `<<nsmulti>>`.

19.3.1 Description

Non-standard multi operations are similar to non-standard operations except for the following restrictions:

- There must be either zero or one parameter;
- The operation must return a struct;
- The SQL for the operation must perform a readmulti.

Typically, this type of operation is intended to be used for doing readmulti operations which join two or more database tables.

This is the only entity operation that cannot utilize additional parameters, which is usually done to provide extra parameters to exit points in the Business Object Layer (BOL). Operations of this stereotype can have either zero or one parameter only. You cannot add any extra parameters to this operation.

19.3.2 Use

You should use *non-standard multiple* operations to either:

- retrieve attributes from multiple database tables, performing a relational join across the tables, or;
- retrieve attributes from one or more database tables when the selection criteria is too complex to use a `<<readmulti>>` or `<<nsreadmulti>>`. For example, if the `where` clause contains comparisons other than equals such as less-than, greater-than, etc.

A non-standard multiple operation is very similar (from a modeling perspective) to a non-standard multiple read operation (`<<nsreadmulti>>`). The major difference is that the designer must specify the SQL to be executed. This enables multiple database tables to be referenced and/or complex `where` clauses to be specified.

19.3.3 Parameter and Generator Notes

- *Parameters* - key parameter [optional].
- *Return value* - list-details parameter.
- *Generator action* - The generator will create a list wrapper for the return-value struct specified by the developer, and use this as the return value for the operation.

You cannot specify any extra parameters for this operation.

19.3.4 Example 1 - nsmulti with a Single (List) Parameter

Consider an operation in the input meta-model to list every single transaction in the system whose amount was for less than one dollar.

The following struct is defined in the model and will be used to contain the information about each transaction. The type of each attribute of the struct is not relevant here and has been omitted for clarity.

- `<<struct>>` class: `MinorTxDetails`

Attribute	Domain
txDate	DATE
txAccountNumber	ACCOUNT_NUMBER

Attribute	Domain
txAmount	AMOUNT

The table below shows an entity defined in the model with some of the attributes, which will be used by the `<<nsmulti>>` operation `getMinorTransactions()`, returning an instance of `MinorTxDetails`.

- `<<entity>>` class: `BankAccount`

Attribute	Domain
`<<details>>` txDate	DATE
`<<details>>` txAccountNumber	ACCOUNT_NUMBER
`<<details>>` txAmount	AMOUNT
`<<details>>` txTellerNumber	TELLER_NUMBER

The SQL for the operation (which must be supplied in the model by the developer) is as follows in Example 19.1, *SQL for nsmulti with a single (list) parameter*:

```
SELECT txAccountNumber, txDate, txAmount
INTO
  :txAccountNumber,
  :txDate,
  :txAmount
FROM    BankAccount
WHERE   txAmount < 1;
```

Example 19.1 SQL for nsmulti with a single (list) parameter

This is all that has to be provided by the developer, the remainder is produced by the generator and is shown below for illustrative purposes.

The following pseudo code in Example 19.2, *Pseudocode for generated structs for use by nsmulti operation* describes the structs used in this operation. The actual *Java* structs corresponding to the structs defined in the model are produced by the code generator.

```
struct MinorTxDetails {
  txDate;
  txAccountNumber;
  txAmount;
};

// this is a generated list wrapper:
struct MinorTxDetailsList {
  sequence <MinorTxDetails> dtls;
};

// this is the standard details struct for the entity
// just to show where its attributes are kept:
struct BankAccountDtls {
  txAccountNumber;
  txDate;
  txAmount;
  txTellerNumber;
}
```

Example 19.2 Pseudocode for generated structs for use by nsmulti operation

The *Java* interface for this entity class - complete with the nsmulti operation is produced by the code generator and would look like this:

```
public interface BankAccount {

  // This is our "nsmulti" operation. Note how the
  // generator has transformed the parameter of this function
  // from "MinorTxDetails" to a "MinorTxDetailsList"
  public MinorTxDetailsList getMinorTransactions()
    throws AppException, InformationalException;
};
```

Example 19.3 Generated Java interface for nsmulti operation

Example 19.4, *Calling a nsmulti operation from handcrafted Java code (one parameter)* demonstrates how the developer would write handcrafted *Java* code to call this method and to iterate through each element returned by the method:

```
<ProjectPackage>.intf.BankAccount bankAccount
  = <ProjectPackage>.fact.BankAccountFactory.newInstance()

double theTotalAmount = 0;

// Call the operation:
MinorTxDetailsList txList
  = bankAccount.getMinorTransactions();

// iterate through the set of results.
for (int i = 0; i < txList.dtls.size(); i++) {
  MinorTxDetails currentTx = txList.dtls.item(i);

  theTotalAmount += currentTx.txAmount;
}
```

Example 19.4 Calling a **<<nsmulti>>** operation from handcrafted *Java* code (one parameter)

19.3.5 Example 2 - nsmulti with Two Parameters (Key + List)

For this example, we will slightly modify the functionality of the previous example.

Instead of returning all transactions for less than one dollar, in the whole system, it will return only the transactions *for one account* which were less than one dollar.

Another parameter is required to specify the account number we are interested in. Since <<nsmulti>> is a database operation and database operations require all parameters to be structs, we must use a struct for our account number parameter even though the struct will have only one field.

Note that the account number field appears in various guises - txAccountNumber, txAccountNum, theAccountID. Unlike the other database opera-

tions, the names of attributes do not have to correspond when used in <<ns>> or <<nsmulti>> operations, the handcrafted SQL can reference the different field names as appropriate.

- <<struct>> class: AccountNoWrapper

Attribute	Domain
txAccountNumber	ACCOUNT_NUMBER

This struct can now be used as an input argument to the <<nsmulti>> operation getMinorTransactions(theAccountID : AccountNoWrapper), returning an instance of MinorTxDetails for the entity below:

- <<entity>> class: BankAccount

Attribute	Domain
<<details>> txDate	DATE
<<details>> txAccountNumber	ACCOUNT_NUMBER
<<details>> txAmount	AMOUNT
<<details>> txTellerNumber	TELLER_NUMBER

The SQL for the operation is shown in Example 19.5, *SQL for nsmulti with a key and list parameters*:

```
SELECT txAccountNumber, txDate, txAmount
INTO
  :txAccountNumber,
  :txDate,
  :txAmount
FROM    BankAccount
WHERE   (txAmount < 1)
  AND   (txAccountNumber = :txAccountNum);
```

Example 19.5 SQL for nsmulti with a key and list parameters

This is all that has to be provided by the developer, the remainder is produced by the generator and is shown below for illustrative purposes.

The following pseudo-code Example 19.6, *Pseudocode for generated structs for use by nsmulti with key and list parameters* describes the structs used in this operation (The actual *Java* structs corresponding to the structs defined in the model are produced by the code generator.):

```
struct MinorTxDetails {
  txDate;
  txAccountNumber;
  txAmount;
};

// this is a generated list wrapper:
struct MinorTxDetailsList {
  sequence <MinorTxDetails> dtls;
};
```

```
struct AccountNoWrapper {
  txAccountNum;
}

// this is the standard details struct for the entity
// just to show where its attributes are kept:
struct BankAccountDtls {
  txAccountNumber;
  txDate;
  txAmount;
  txTellerNumber;
}
```

Example 19.6 Pseudocode for generated structs for use by nsmulti with key and list parameters

The *Java* interface for this entity class - complete with the nsmulti operation - is produced by the code generator and would look like the following:

```
public interface BankAccount {

  // This is our "nsmulti" operation. Note how the
  // generator has transformed the return value of this
  // function from "MinorTxDetails" to a
  // "MinorTxDetails_List"
  public MinorTxDetailsList getMinorTransactions
                              (AccountNoWrapper   theAccountID)
    throws AppException, InformationalException;
};
```

Example 19.7 Generated Java interface for nsmulti operation with key and list parameters

Example 19.8, *Calling a nsmulti operation from handcrafted Java code (two parameters)* demonstrates how the developer would write handcrafted *Java* code to call this method and to iterate through each element returned by the method:

```
<ProjectPackage>.intf.BankAccount bankAccount
  = <ProjectPackage>.fact.BankAccountFactory.newInstance();

AccountNoWrapper accNoWrapper = new AccountNoWrapper;

accNoWrapper.txAccountNum = "57033186";

double theTotalAmount = 0;

// Call the operation:
MinorTxDetailsList  txList
  = bankAccount.getMinorTransactions(accNoWrapper);

// iterate through the set of results.
for (int i = 0; i < txList.dtls.size(); i++) {
  MinorTxDetails currentTx = txList.dtls.item(i);

  theTotalAmount += currentTx.txAmount;
}
```

Example 19.8 Calling a **<<nsmulti>>** operation from handcrafted *Java* code (two parameters)

19.4 Using Handcrafted SQL in Non-Standard Entity Operations

19.4.1 Overview

For entity operations of stereotype `<<ns>>` and `<<nsmulti>>` the developer is required to specify the SQL to be used in the Cúram Data Access Layer (DAL).

These queries have access to all the tables on the database and to all the parameters of the operation.

19.4.2 Using Host Variables

Host variables in SQL directly reference fields in the parameter struct or return value struct.

The rules for using host variables are as follows:

- host variables must be prefixed with a colon (:);
- host variables are case sensitive.

For example:

`:surname`

- if a field in the parameter struct or return value struct is a result of aggregation then the role name of aggregation is used for host variable.

For example:

`:dtls` (see Section 20.4, *One-to-One Aggregation*)

19.4.3 "Null" Considerations

When writing a handcrafted SQL statement, it is important to note that some Cúram datatypes are stored as `null` on the database if they are empty (i.e. in their initial state), so when searching for these records your query must search for "`null`" rather than an empty string. For example:

Incorrect

```
SELECT ... INTO ... FROM ... WHERE someStringColumn = '';
```

Correct

```
SELECT ... INTO ... FROM ....WHERE someStringColumn is null;
```

In general, if the Cúram data type corresponds to a *Java* class (as opposed to a primitive *Java* type) then its empty state is stored on the database as a `null`. If the data type corresponds to a primitive *Java* type then a `null` on the database is not a valid value for it and the `Allow NULLs on this database field` option defaults to `no`. If necessary this default can be overridden.

Note

The `Allow NULLs on this database field` option controls the `NOT NULL` qualifier in generated DDL in an inverted way. Setting this option to `no` causes the `NOT NULL` qualifier to be added; setting it to `yes` causes the qualifier to be omitted.

The following table shows which Cúram data types can be represented as a `null` on the database.

Datatype	Nulls allowed
SVR_BLOB	yes
SVR_BOOLEAN	no
SVR_CHAR	no
SVR_DATE	yes
SVR_DATETIME	yes
SVR_DOUBLE	no
SVR_FLOAT	no
SVR_INT8	no
SVR_INT16	no
SVR_INT32	no
SVR_INT64	yes
SVR_MONEY	no
SVR_STRING	yes

Table 19.1 Data types and nulls

19.4.4 For Update Considerations With DB2 for z/OS

If running against a *DB2 for z/OS* database, any handcrafted SQL that explicitly uses a `FOR UPDATE` clause may need to be modified to prevent `RecordLockedException` errors from being thrown. If the particular SQL statement is invoked simultaneously by multiple users, you should consider using `FOR UPDATE WITH RS USE AND KEEP UPDATE LOCKS` instead. The locking behavior of *DB2 for z/OS* is subtly different to that of *DB2* on distributed platforms. The `KEEP UPDATE LOCKS` syntax ensures that the locking behavior with *DB2 for z/OS* is the same as it is on distributed platforms.

19.4.5 SQL Example 1

Consider an example where the entity `Employer` has a method `CountEmployers` (stereotype `<<ns>>`) which returns the number of records in the `Employer` table.

The following struct is required to return the result, since stereotyped entity operations cannot return primitive types:

```
public final class LongWrapper
implements Serializable, DeepCloneable {
  /**
   *  LONG_TYPE -> long
   */
  public long longValue = 0;
}
```

Example 19.9 Struct for return result

The *Java* interface for this operation would look like the following extract:

```
public interface Employer
{
  public LongWrapper countEmployers()
    throws AppException, InformationalException;
}
```

Example 19.10 Java Interface

Finally, the SQL to implement this query is:

```
SELECT count(*)
INTO :longValue
FROM Employer;
```

Example 19.11 SQL Implementation

Note that we do not need to specify the name of the `LongWrapper` class, we simply reference the name of the `longValue` attribute within that class because the `INTO` clause is automatically assumed to reference the return value.

Thus if an attribute with the same name is used in the input parameter struct and return value struct then it is assumed that INTO clause references the attribute of return value struct.

19.4.6 SQL Example 2

This example shows how to use parameter host variables and expands the previous example by adding another method which updates a numeric field on one record of the `Employer` table.

```
public interface Employer
{
```

```
  public void setEmployerSize(EmployerKey empKey,
                              LongWrapper newSize)
    throws AppException, InformationalException;
public LongWrapper countEmployers() throws AppException; }
```

Example 19.12 Java Interface

The following struct is required to contain the primary key for the employer:

```
public final class EmployerKey
implements Serializable, DeepCloneable {
  /**
   *  REFERENCE_NUMBER -> String
   */
  public String employerNumber = "";
}
```

Example 19.13 Struct for employer key

The SQL statement for this method is:

```
  UPDATE Employer
  SET size = :2.longValue
  WHERE employerNumber = :employerNumber;
```

Example 19.14 SQL Implementation

Note that since `longValue` is contained in the second parameter it is necessary to qualify it with `2.`. Unqualified parameter references are assumed to reference the first parameter.

The SQL statement below qualifies both parameters and is equivalent to the one above:

```
  UPDATE Employer
  SET size = :2.newSize.longValue
  WHERE employerNumber = :1.employerNumber;
```

Example 19.15 SQL Implementation with qualified parameters

Chapter 20

Aggregation

20.1 Overview

Aggregation is essentially the ability to embed or nest instance(s) of one type of class within another type of class.

The Cúram generator supports two types of aggregation relationships: one-to-one and one-to-many. One-to-one aggregation has the effect of embedding a single instance of one class within another. One-to-many aggregation has the effect of embedding a sequence of one class within another.

The main use for aggregation in the generator is to represent sequences in the input meta-model.

20.2 Rules when Using Aggregation

The generator permits the following aggregation configurations:

- structs can aggregate structs;
- structs can aggregate entities.

20.3 A Special Case

The generator supports the aggregating of standard details structs, even though they do not appear in the input model. Standard details structs are aggregated by aggregating the entity class which “owns” the standard details struct.

20.4 One-to-One Aggregation

The following example describes how to aggregate a struct class, `Person-`

Details, into to another struct class, PersonDetailsWrapper, using one-to-one aggregation.

To create a one-to-one aggregation create an *Rational Software Architect* diagram and do the following:

- Add classes PersonDetails and PersonDetailsWrapper to the diagram;
- In the diagram drag the appropriate arrowhead (appears when the mouse cursor is over the class) between the two classes with PersonDetailsWrapper set as the source and PersonDetails the target;
- Select Create Aggregation from the popup menu;
- With the aggregation relationship selected in the diagram open the General Properties tab.

This creates the aggregation relationship whereby one role corresponds to class PersonDetailsWrapper and the other to class PersonDetails. A UML role is essentially one end of a UML relationship so each relationship has two roles whose names are Role A and Role B. Exactly one of these roles - usually Role A - will have its Aggregate option set. The assignment of Role A and Role B is arbitrary. The key thing to remember is that the role which has the Aggregate box checked denotes the outermost class of the pair.

With the relationship line selected in the diagram the General Properties tab should show PersonDetailsWrapper in the graphic at the top of the properties sheet with the diamond associated with it. Set the following properties of the aggregation:

- The Label is optional;
- For PersonDetailsWrapper:
  - The Aggregation radio button should indicate Composite;
  - Multiplicity should be set to 1;
- For PersonDetails:
  - The Aggregation radio button should indicate None;
  - By default the role is set to "dtls";
  - Multiplicity should be set to 1 (to signify a one-to-one aggregation).

The class diagram would appear in the *Rational Software Architect* showing the two classes joined by the UML aggregation relationship line (diamond end touching PersonDetailsWrapper) and each side of the relationship showing multiplicity of one and the PersonDetails showing a role name of - dtls.

Note

The position of the diamond in the model diagram is important as it denotes the outermost class in the pair.

The generated *Java* code resulting from this construct would take the following format:

```
public final class PersonDetails implements
java.io.Serializable, curam.util.type.DeepCloneable {
  public String personRefNo = "";
  public String firstName = "";
};
public final class PersonDetailsWrapper implements
java.io.Serializable, curam.util.type.DeepCloneable {
  // This class has a single instance of
// class "PersonDetails" embedded in it. PersonDetails dtls =
//   new PersonDetails();
};
```

20.5 One-to-Many Aggregation

In this example a one-to-many aggregation is modeled, meaning that a list of one class type is embedded into the other class. Here we create `PersonDetailsList`, which aggregates a list of `PersonDetails`. To create a one-to-many aggregation, open an *Rational Software Architect* diagram and do the following:

- Add classes PersonDetails and PersonDetailsList to the diagram;
- In the diagram drag the appropriate arrowhead (it appears when the mouse cursor is hovering over the class) between the two classes with PersonDetailsList as the source and PersonDetails as the target;
- Select Create Aggregation from the popup menu;
- With the aggregation relationship selected in the diagram open the General Properties tab.

This creates the aggregation relationship whereby one role corresponds to class PersonDetailsList and the other to class PersonDetails.

Note

The position of the diamond is important as it denotes the outermost class in the pair.

With the relationship line selected in the diagram the General Properties tab should show PersonDetailsList in the graphic at the top of the properties sheet with the diamond associated with it.

Set the following properties of the aggregation:

- The Label is optional;
- For PersonDetailsList:

    - The Aggregation radio button should indicate Composite;
    - Multiplicity should be set to *;
- For PersonDetails:
    - The Aggregation radio button should indicate None;
    - By default the role is set to "dtls";
    - Multiplicity should be set to 1..* (to signify a one-to-many aggregation).

The class diagram would appear in the *Rational Software Architect* showing the two classes joined by the UML aggregation relationship line (diamond end touching `PersonDetailsList`) and the aggregates side of the relationship showing a multiplicity of `*` and `PersonDetails` showing a multiplicity of `1..*` and a role name of `- dtls`.

The pseudo-code resulting from this construct would take the following format:

```
struct PersonDetails implements
java.io.Serializable, curam.util.type.DeepCloneable {

  String personRefNo = "";
  String firstName = "";
};

struct PersonDetailsList implements
java.io.Serializable, curam.util.type.DeepCloneable {

  public static class List_dtls
  extends curam.util.type.ValueList {
    public void addRef(PersonDetails s) {
      add(s);
    }
    public PersonDetails item(int indx) {
      return (PersonDetails) get(indx);
    }
    public PersonDetails[] items() {
      PersonDetails[] result = new PersonDetails[size()];
      toArray(result);
      return result;
    }
  }

  // This class contains an embedded list of "PersonDetails":
  public final List_dtls dtls = new List_dtls();

}
```

The resulting generated struct class for `PersonDetailsList` has a field named `dtls` which provides functionality required for lists such as adding items, getting an item by index and getting the list contents as an array.

Chapter 21

Assignable

21.1 Overview

A function of the generated `<<struct>>` class is the ability to automatically assign values between matching fields in another `<<struct>>` as provided by the generated `<<struct>>` class's super class `curam.util.type.struct.Struct`. Consider an example of a `<<struct>>`, `BankBranchStruct` with several attributes:

- `bankBranchID`
- `bankId`
- `bankName`
- `bankSortCode`
- `name`
- etc.

A `BankBranchListDetails` `<<struct>>` class has a subset of attributes shared with the `BankBranchStruct` class:

- `bankBranchID`
- `bankSortCode`
- `name`

Based on this example modeling the following *Java* code illustrates how to create these objects.

```
BankBranchStruct bankBranchStruct
  = new BankBranchStruct();
BankBranchListDetails bankBranchListDetails
  = new BankBranchListDetails();
```

Typically, the assignment from one struct to the other might look like this:

```
bankBranchListDetails.bankBranchID
        = bankBranchStruct.bankBranchID;
bankBranchListDetails.bankSortCode
        = bankBranchStruct.bankSortCode;
bankBranchListDetails.name = bankBranchStruct.name;
```

The above code can be simplified as follows using the `assign` function, which becomes more significant as the size of the structs increases:

```
bankBranchListDetails.assign(bankBranchStruct);
```

An `<<assignable>>` relationship then is one which allows further control of the specifics of the automatic assignment with the `assign` function. It is required where you want to do explicit field assignment between fields with differing names or to suppress the default assignment between fields of the same name.

21.2 Explicit Field Assignment

An explicit field assignment is one where fields with different names are matched. It is represented in the model by adding an `assignable` relationship between the two classes, and then adding attributes to be matched to the both sides of the assignment. Any fields which are not explicitly linked will be treated as default assignment fields.

The following classes are used to illustrate this.

- `<<entity>>` class: `Address`

Attribute
addressID
addressLine1
addressLine2
addressLine3
addressLine4
cityCode
countryCode
postalCode
regionCode
comments

- `<<struct>>` class: `BankBranchStruct`

Attribute
bankBranchID

Attribute
bankID
bankName
addressID
addressLine1
addressLine2
addressLine3
addressLine4
countryCode
postalCode
regionCode
addressVersionNo
cityID

In an `assignable` relationship between the two classes `Address` and `BankBranchStruct` fields can be explicitly mapped; e.g. `BankBranchStruct.cityID` matched with `Address.cityCode`. In *Rational Software Architect* this is shown in *Role:* fields (RoleA & RoleB) of the *General* tab of the assignable relationship with the linked pair, `cityID` in one Role field and `cityCode` in the other. All the other common fields (e.g. `AddressLine1`, etc.) are handled automatically by the generator.

For instance, the generated code without the explicit field assignment would appear as shown below:

```
public curam.util.testmodel.struct.BankBranchStruct
    assign(final curam.util.testmodel.struct.AddressDtls v)
{
  addressID = v.addressID;
  addressLine1 = v.addressLine1;
  addressLine2 = v.addressLine2;
  addressLine3 = v.addressLine3;
  addressLine4 = v.addressLine4;
  countryCode = v.countryCode;
  postalCode = v.postalCode;
  regionCode = v.regionCode;
  return this;
}
```

With the explicit field assignment the following code is then added to the `assign` method: `cityID = v.cityCode`. The handcrafted *Java* to assign these structures would be as follows:

```
BankBranchStruct dtls = new BankBranchStruct();
AddressDtls addressDtls = new AddressDtls();
dtls.addressLine1 = addressDtls.addressLine1;
dtls.addressLine2 = addressDtls.addressLine2;
dtls.addressLine3 = addressDtls.addressLine3;
dtls.addressLine4 = addressDtls.addressLine4;
dtls.cityID = addressDtls.cityCode;
dtls.countryCode = addressDtls.countryCode;
dtls.postalCode = addressDtls.postalCode;
dtls.regionCode = addressDtls.regionCode;
```

By using the generated assignment operator, these lines of code can be re-

duced to just one line as follows:

```
bankDtls.assign(addressDtls);
```

21.3 Suppressing Default Assignment Fields

In some situations you may not want a pair of similarly named fields to be matched. You can cause a pair of fields to be omitted from an assignment by listing one of the fields at one end of the relationship.

For the following two classes below, `PersonInfo` and `AccountInfo`, having an `<<struct>>` relationship, the same named fields are matched.

- `<<struct>>` class: `AccountInfo`

Attribute
Id
Surname
FirstName
Balance

- `<<struct>>` class: `PersonInfo`

Attribute
Id
Surname
FirstName

For this example we first create the objects for the `PersonInfo` and `AccountInfo` classes as described above:

```
AccountInfo account = new AccountInfo();
PersonInfo person = new PersonInfo();
```

This assignment:

```
account.assign(person);
```

is equivalent to the following three statements:

```
account.Id = person.Id;
account.Surname = person.Surname;
account.FirstName = person.FirstName;
```

By adding `Id` as a key to one end of the relationship, it is excluded from the generated assignment and now this assignment:

```
account.assign(person);
```

is equivalent to the following *two* statements; that is, the `Id` assignment will no longer be made:

```
account.Surname = person.Surname;
account.FirstName = person.FirstName;
```

21.4 Combining structs

Sometimes you may need to populate one struct with the contents of two or more other structs.

A typical piece of *Java* code would look like the following:

```
BankBranchStruct dtls = new BankBranchStruct();
AddressDtls      addressDtls = new AddressDtls();
BankBranchDtls   bankBranchDtls = new BankBranchDtls();

// Copy from the "AddressDtls" struct
dtls.addressLine1 = addressDtls.addressLine1;
dtls.addressLine2 = addressDtls.addressLine2;
dtls.addressLine3 = addressDtls.addressLine3;
dtls.addressLine4 = addressDtls.addressLine4;
dtls.cityCode     = addressDtls.cityCode;
dtls.countryCode  = addressDtls.countryCode;
dtls.postalCode   = addressDtls.postalCode;
dtls.regionCode   = addressDtls.regionCode;
dtls.addressVersionNo = addressDtls.versionNo;

// Copy from the "BankBranchDtls" struct
dtls.bankBranchID = bankBranchDtls.bankBranchID;
dtls.bankID       = bankBranchDtls.bankID;
dtls.bankSortCode = bankBranchDtls.bankSortCode;
dtls.name         = bankBranchDtls.name;
dtls.versionNo    = bankBranchDtls.versionNo;
```

Example 21.1 Example Java code for combining structs

By explicitly mapping the `BankBranchStruct.addressVersionNo` attribute to the `Address.versionNo` in the `<<assignable>>` relationship the *Java* can now be written as:

```
// Copy from the "AddressDtls" struct
dtls.assign(addressDtls);

// Copy from the "BankBranchDtls" struct
dtls.assign(bankBranchDtls);
```

Example 21.2 Equivalent Java code for combining structs

Note that in this case, the second `assign` does not overwrite the first as it happens to reference a different subset of fields, so the net effect is that the two struct contents are merged.

Chapter 22

Foreign Keys

22.1 Overview

The Cúram generator allows for foreign keys to be created between database tables.

A *Foreign Key* relationship between two database tables is specified in the input model by adding a relationship of stereotype `<<foreignkey>>` (one word, no spaces) between two entity classes. Optionally you can give the relationship a name, this name is then applied to the foreign key constraint added to the database. Otherwise the database chooses its own name for the constraint.

22.2 Rules when Using Foreign Keys

- Foreign key relationships are allowed on entity classes only.
- Fields referenced by a foreign key will be set to unique, as this is required by some databases.
- If the foreign key references the *primary* key of another entity, a redundant `unique` clause will not be generated by the generator, as the primary key is already unique.
- Foreign keys cannot be specified on subclass entities. The relationship should be specified using the actual base entity classes themselves.

22.3 How to Add a Foreign Key to a Database Table

A foreign key is specified between a pair of entities by adding a relationship between the two classes and adding key/qualifiers to the role touching the *referenced* class. On a class diagram, this results in a line between two

classes, with a box containing the key/qualifiers at the *referencing* class.

The notation for linking pairs of fields in two different classes is the same for foreign keys as for generated assignments. The class diagram will show two classes joined by a line with pairs of linked attributes in a box at one end of the line. The first name in the pair refers to an attribute in the nearer class, the second name refers to an attribute in the other class.

22.4 Naming Primary and Foreign Key Constraints

It is possible to include a constraint name for foreign key constraints in Cúram models. The name given in the model to the `<<foreignkey>>` relationship will be applied to the foreign key constraint itself. If necessary this feature can be suppressed by specifying '`-nonamedforeignkeyconstraint`' on the generator command line.

Primary key constraints are also given names in the database. The name of each constraint is the same as that of its corresponding entity. This also results in an accompanying index of the same name. This feature can be suppressed by specifying `-nonamedprimarykeyconstraint` on the generator command line.

22.5 Example

Consider two `<<entity>>` classes, `BankAccount` and `BankTransaction`, where `BankAccount.accountNo` is a `<<foreignkey>>` on `BankTransaction`. That is, the BankTransaction table (txAccountNo) must have a record on the BankAccount table with a matching accountNo value.

The tables below illustrate these two classes where the `<<foreignkey>>` would be between their `<<key>>` attributes:

- `<<entity>>` class: `BankAccount`

Attribute	Domain
`<<key>>`*accountNo*	ACCOUNT_NO
`<<details>>` clientID	CLIENT_ID
`<<details>>` branchLocation	BRANCH_LOCATION
`<<details>>` currentBalance	MONEY
`<<details>>` lastTransaction	DATE_TYPE
`<<details>>` lastStatement	DATE_TYPE

- `<<entity>>` class: `BankTransaction`

Attribute	Domain
`<<key>>`*txAccountNo*	ACCOUNT_NO

Attribute	Domain
<<details>> txID	TX_ID
<<details>> transactionDate	TX_DATE
<<details>> transactionType	TX_TYPE
<<details>> transactionAmount	TX_AMOUNT

This foreign key results the following DDL being generated (*Oracle* SQL shown):

```
ALTER TABLE BankTransaction ADD(
  FOREIGN KEY (txAccountNo)
  REFERENCES BankAccount(accountNo));
```

Chapter 23

Indices

23.1 Overview

The Cúram generator allows for indices other than the primary index to be created on database tables. Any number of indices can be created on each table, with the usual speed vs. database size trade-offs associated with indices.

An index for a database table is specified in the input model by adding a relationship of stereotype `<<index>>` between an entity class and a struct class.

The fact that a struct is being used to represent an index does not have any side-effects on the struct apart from those mentioned in the rules below, i.e. the struct can still be used as an argument to an operation. Typically the struct would be used as both a key parameter and as an index to support database accesses via this key.

23.2 Rules when Using Indices

- The relationship must be given a name. This is the name which will be given to the database index. (The name of the struct in the relationship does not have any effect on the index.)
- The names of the attributes of the struct class must be a subset of the names of the attributes of the entity.
- The struct class must not aggregate any other classes.
- Index names must be unique within the entire model.

23.3 How to Add an Index to a Database Table

Create a struct class whose fields are a subset of the fields of the entity class.

Add a relationship of stereotype <<index>> between the entity class and the struct. The direction of the relationship is not important.

Set the relationship name to the name which you want given to the database index.

23.4 Naming Indices

Developers will never explicitly reference an index but the DBA[1] will, so it is recommended that index names be kept as meaningful and descriptive as possible.

23.5 Example

Consider the following two classes with an <<index>> relationship named BankClientMNIndex:

- <<entity>> class: BankClient

Attribute	Domain
<<key>> clientID	CLIENT_ID
<<details>> firstName	CUSTOMER_NAME
<<details>> middleName	CUSTOMER_NAME
<<details>> lastName	CUSTOMER_NAME
<<details>> address1	ADDRESS_LINE
<<details>> address2	ADDRESS_LINE
<<details>> address3	ADDRESS_LINE
<<details>> address4	ADDRESS_LINE

- <<struct>> class: MiddleNameWrapper

Attribute	Domain
middleName	CUSTOMER_NAME

The above index results in the following DDL being produced by the generator:

```
CREATE INDEX BankClientMNIndex
ON BankClient(middleName);
```

Notes

[1]Data Base Administrator.

Chapter 24

Unique Indices

24.1 Overview

A unique index in a *IBM Cúram Social Program Management* model is modeled by adding a relationship of stereotype `<<uniqueindex>>` between an entity class and a struct class. The rules for modeling a unique index are the same as those for modeling a non-unique index.

Specifying a unique index for an entity causes the necessary information to be included in the generated file `<Application-name>_unique_constraints.xml` which must then be referenced from the data manager configuration file (`datamanager_config.xml`).

Note that the file `<Application-name>_unique_constraints.xml` contains two sets of information:

1. Unique indexes. These correspond to explicit '`<<uniqueindex>>`' relationships in the model and result in DDL of the form:

```
CREATE UNIQUE INDEX <index-name>
        ON....
```

   where '`<index-name>`' is the name of the relationship in the model.

2. Unique constraints. These are implicit unique constraints which are produced automatically by the generator and which are applied to all fields which are referenced by a foreign key. They correspond to `<<foreignkey>>` relationships in the model and result in DDL of the form:

```
ALTER TABLE <table-name> ADD
        UNIQUE...
```

   or, if there is a `<<uniqueindex>>` for the fields referenced by the

foreign key:

```
ALTER TABLE
        <table-name> ADD CONSTRAINT <constraint-name>
        UNIQUE...
```

where '<constraint-name>' is the name of the corresponding <<uniqueindex>> relationship in the model.

When the data manager is run, the explicit unique indexes are created before the implicit unique constraints. This allows the database to use the developer-specified unique indexes to enforce uniqueness rather than having to create and use its own system-named indexes. For example the developer may wish to model their specifically named unique index to correspond to a particular foreign key in the model. In this case the generator will automatically give the unique constraint the same name as the corresponding unique index.

Chapter 25

Generated Class Hierarchy

25.1 Overview

This section describes the hierarchy of classes generated by the server code generator, and shows how they correspond to the classes designed in the application model.

All classes are defined in the *IBM Cúram Social Program Management* model using UML notation. A single process, facade or entity class may contain a mixture of automatically generated methods, and methods that the application developer is required to implement. It is not desirable to store handcrafted code and generated code in the same file due to the risk of the generator overwriting handcrafted code, or vice versa. Therefore all developer code is stored in a single class, generated code is produced into a number of other classes, and the set is linked together into a hierarchy by inheritance and implementation.

Note that since struct classes do not contain operations there are no issues of separating handcrafted and generated code. Therefore each struct class in the model corresponds to one generated *Java* struct class.

25.2 Basic Hierarchy Example

This section describes the elements of the generated and required handcrafted class hierarchy for a basic entity class named `MyClass`, which does not make use of inheritance or code packages.

The UML representation of the generated *Java* classes of the `<<entity>>` class `MyClass` would show the following four classes:

- `<PackageName>.intf.MyClass`
- `<PackageName>.base.MyClass`

    - Implements, or realizes, the `intf` class.
    - It is the super class.
- `<PackageName>.impl.MyClass`
    - A subclass of the `base` class.
    - Contains any required (non-generated) handcrafted methods.
- `<PackageName>.fact.MyClassFactory`
    - A subclass of the `impl` class.
    - Returns an instance of the `intf` class.

Thus, there are four *Java* classes corresponding to the `<<entity>>` class in the UML model. Three of the classes have the same name as the class in the model, the fourth has the same name with the word `Factory` appended.

A further description of the classes are as follows:

1. `<ProjectName>.intf.MyClass`

   This is a generated *Java* interface class containing all the public methods for the class.

   The other classes in the hierarchy - either generated or handcrafted - will be required to provide implementations for these methods.

2. `<ProjectName>.base.MyClass`

   This is a generated abstract *Java* class which implements the interface contained in the `intf` version of the file. It contains the following:

    - The implementations of data access methods (i.e. stereotyped methods of entity classes) and connector methods.
    - Abstract method declarations for exit point methods.

      This is to ensure that the developer is forced to provide implementations for the exit points.
    - Abstract method declarations for methods declared protected in the model.

      This is to ensure that the developer is forced to provide implementations for these methods without having to expose them in the interface (`intf` layer) for the class.

3. `<ProjectName>.impl.MyClass`

   This class is supplied by the developer and always inherits from the corresponding `base` version.

   It should be declared abstract to ensure that the class cannot be instanti-

ated directly - the class should only be instantiated using the factory mechanism. (See below.)

In this class the developer must provide implementations for all the methods declared in the class in the model for which an implementation was not produced by the generator.

While this class inherits from a generated class, it contains only handcrafted code and *no* generated code. This is so that there is no risk of developer code overwriting generated code, or generated code overwriting developer code.

4. `<ProjectName>.fact.MyClassFactory`

   This is a generated *Java* class containing one static method: `newInstance()`. This method creates instances of the class and is the only means by which entity, facade and process classes should be instantiated.

   Since a factory creates all instances of objects, it can also be used to:

   - transparently create and return a customized version of the class requested. See Section 27.3.2, *Replacing the Superclass*. Pre-existing code which used the original version of the class does not need to be changed.
   - transparently create and return a proxy class of the requested class. The proxy class wraps the requested class (using the *Java* 1.3 Dynamic Proxy mechanism) and captures detailed tracing information for all interactions with the class.

The following code sample shows how an instance of `MyClass` is created. Note that the return type of `MyClassFactory.newInstance` is `sample.intf.MyClass`.

```
// Use the factory to create an instance:
sample.intf.MyClass myObject =
        sample.fact.MyClassFactory.newInstance();
```

Example 25.1 Using a factory to create an instance of MyClass

25.3 Hierarchy for Subclasses

This section describes the elements of the generated/handcrafted class hierarchy for a basic entity class named `SubClass` that inherits from `MyClass`.

The UML representation of the generated *Java* classes for the `<<entity>>` class `SubClass` would show the following four classes:

- `<PackageName>.intf.SubClass`
  - It inherits from the `MyClass` `intf` class.

- `<PackageName>.base.SubClass`
  - Implements, or realizes, the `intf` class.
  - It is the super class.
  - It inherits from the `MyClass impl` class.
- `<PackageName>.impl.SubClass`
  - A subclass of the `base` class.
  - Contains any required (non-generated) handcrafted methods.
- `<PackageName>.fact.SubClassFactory`
  - A subclass of the `impl` class.
  - Returns an instance of the `intf` class.

As with the previous example there are four *Java* classes corresponding to class `SubClass`. However the fact that `SubClass` inherits from `MyClass` results in two additional relationships, highlighted here:

1. Interface `SubClass` inherits from interface `MyClass` thereby ensuring that `SubClass` must implement all of its own declared methods plus those declared in `MyClass`.
2. Generated class `<ProjectName>.base.SubClass` inherits from handcrafted class `<ProjectName>.impl.MyClass`. This means that `SubClass` inherits the implementations of the methods from `SubClass` as well as their declarations, so these methods are available to `SubClass` and do not have to be re-implemented.

25.4 Hierarchy for Abstract Classes

In a Cúram model the developer can mark classes `abstract` (See Section 6.6, *Options*) meaning that they cannot be instantiated.

From the above example, if `MyClass` were qualified `abstract`, the following hierarchy would result:

- `<PackageName>.intf.MyClass`
- `<PackageName>.base.MyClass`
  - Implements, or realizes, the `intf` class.
  - It is the super class.
- `<PackageName>.impl.MyClass`
  - A subclass of the `base` class.

- Contains any required (non-generated) handcrafted methods.

The hierarchy is the same as for non-abstract classes except that no factory is generated.

25.5 Considerations

25.5.1 Access Control - private/protected/public/package

The *Java* language supports four levels of access control for methods and member variables. In Cúram models this is simplified to two levels: public and protected. Since the generated class hierarchy includes different classes in different packages it does not make sense to use the private and package access levels. Note that this applies only to operations in Cúram models - developers are still free to use private and public access in handcrafted *Java* code as they see fit.

25.5.2 The Meaning of **super**

In *Java*, the super keyword is a reference to the superclass i.e. the class from which the current class (`this`) inherits.

In the example shown in Section 25.3, *Hierarchy for Subclasses* the superclass of `SubClass` is `MyClass`. However when writing handcrafted *Java* code for `<ProjectName>.impl.SubClass` it is important to remember that the superclass of this class is actually `<ProjectName>.base.SubClass` rather than any version of `MyClass`.

25.5.3 Enforcing the Factory Mechanism

For reasons mentioned above, entity, facade and process objects should be created only by using their associated factory classes. Developers should not bypass this mechanism by using the `new` keyword to instantiate these classes. This can and should be enforced by making all implementation classes (i.e. all classes in the `impl` packages) abstract. Failure to make these classes abstract means that there is a risk of developers instantiating them directly with the result that class replacement will not work as expected.

25.6 Summary

Certain individual objects in a Cúram application model appear as multiple classes in the output code. The objective of the generated class hierarchy is to ensure the following:

- The developer provides all handcrafted implementation within a single *Java* class.

- The public parts of the object's interface are accessible to other objects and the non-public parts of the object's interface are not accessible to other objects.
- The developer is forced to implement all of the declared interface, both public and non-public - unless the generator produces the necessary implementation.
- Objects can be subclassed and a subclass can be defined to replace its superclass transparently.
- The run time type of an object is determined by a factory, to support replacement and tracing.

Chapter 26

Cúram JMS Queue Connectors

26.1 Overview

IBM Cúram Social Program Management connectors provide a way for a Cúram application to connect to other systems by means of JMS queues. For facade and process class operations with a stereotype of `qconnector` the generator will produce code that converts the operation parameter into a JMS message, places the message on a queue, and optionally waits for another message in response which is then converted back into a Cúram struct and returned to the caller.

For many operations, queue connectors can be implemented without writing any handcrafted code. It is also possible to customize connectors with the use of handcrafted code. You may wish to do this if:

- the default encoding of a datatype is not suitable for your purpose. For example, you may wish to encode dates in the form `DD-MMM-YYYY` instead of the default format of `YYYYMMDD`.
- your parameter struct is “complex”. For example, it may contain a variable length field, or may aggregate another struct.

26.2 How It Works / What It Does

Connections are not created directly, but are built using a connection factory. Factory objects are stored in a JNDI namespace, insulating the JMS application from provider-specific information.

The fields in the parameter struct are scanned using *Java* reflection, and each is converted into a fixed length string based on its datatype. The strings are concatenated together into a JMS `BytesMessage` which is then placed on a JMS queue.

If a return type has been specified for the operation, the Cúram application will wait for a response message, typically on another queue. The remote system must create a correctly formatted response message and send it to the Cúram application within the specified timeout period. When the message is received, it is converted into an instance of the return type struct which is then returned to the caller.

26.3 Options on **<<qconnector>>** Operations

The following options are available on `qconnector` operations:

- **`JNDI name of the QueueConnectionFactory class.`** *Mandatory*. This specifies the name of the QueueConnectionFactory class in the JNDI namespace.

  Queue connections are not instantiated directly but are instead created by connection factories. The connection factories are stored in the JNDI namespace of the application server.

- **`JNDI name of the transmission queue.`** *Mandatory*. This specifies the JNDI name of the queue onto which outgoing messages are placed.

- **`Response message timeout (seconds).`** This is only relevant for operations that have a return value. The return value is obtained by receiving a response JMS message from the recipient and this timeout value is used to ensure that the application does not wait indefinitely for the response.

  Default value: 30 seconds.

- **JNDI name of the reply queue.** This is only relevant for operations that have a return value. Specifies the JNDI name of the queue from which the response message should be taken.

- **Message Type.** BytesMessage or TextMessage. This allows you to specify whether a JMS `BytesMessage` or `TextMessage` is sent/received by the connector. By default the JMS connectors send and receive a JMS `BytesMessage` containing the bytes of a string representation of the struct parameters. If the system(s) being communicated with use a different character encoding, then these bytes may not be correctly translated by the other systems. In this case - provided the message doesn't contain any binary data - a JMS `TextMessage` can be used to ensure that the message is correctly translated by the other systems.

- **BytesMessage encoding character set.** Specifies the name of the character encoding to use when converting the string representation of a struct to a JMS `BytesMessage`, and vice versa. If this option is not specified then the default local system character encoding is used. (Usually 'Cp1252' for *Microsoft® Windows*, 'Cp1046' for EBCDIC on *IBM® z/OS®*, etc.) This enables you to ensure that the character encod-

ing used for the message matches the character encoding of the other system being communicated with.

This option is not relevant if TextMessage is used.

26.4 How to Use <<qconnector>> Operations

The following section explains how qconnector operations are represented in the meta-model and implemented on the remote system.

26.4.1 Decide on Format of Message and Create the struct(s) to Correspond to the Message

The Cúram developer and the remote application developer need to agree on the format(s) of the messages passed between the two systems. This involves:

- **The format of each field in the message.** The default encoding method can be used for each field; but, see Table 26.1, *Encoding methods* and Section 26.7, *Using Customized Encoding/Decoding Classes* for how a custom encoding methodology may be implemented.
- **The length of each field in the message.** Like the encoding, the encoded length of each field depends on the type of the field and - for some datatypes - its length as specified in the model. See Table 26.1, *Encoding methods* for information on lengths of datatypes. The length of the field can be changed by implementing a custom mapper for the field.
- **The ordering of the fields in the message.** The fields appear in the message in the same order as they appear in the struct in the meta-model. The tool-bar contains a facility for changing the order of struct attributes if required.

26.4.2 Add the operation to the application meta-model.

A qconnector operation is modelled like any other process or facade class operation subject to the restrictions listed in Section 26.5, *Rules / Restrictions*. It is also necessary to use some of the operations listed in Section 26.3, *Options on qconnector Operations* to specify the queue(s) and some queuing parameters.

In summary, the method should have one struct parameter, may return void or a struct, and should have options set to identify the *MQSeries* queues to use.

26.4.3 Configure the Queues in the Application Server

The queue connection factory, and references to the queues themselves are stored in the JNDI namespace. These JNDI names are be mapped to actual

connection factories and queues in the application server configuration.

26.4.4 Implement the message recipient in the remote system

The message recipient can be any system which has access to the *MQSeries* queues. Typically this will be a legacy system to which access is required by the Cúram application. The target system can be either a JMS application or a basic *MQSeries* application.

If no response is required from the remote system, the remote system simply collects and decodes the received message, and uses it as required.

If a response message is required, i.e. if a return type has been specified for the operation, then the remote system must create a response message and send it back to the waiting Cúram application. The response message is associated with the original message using its `CorrelationID`, i.e., *the message recipient must set the* `CorrelationID` *of the response message equal to the* `MessageID` *of the original message.*

26.5 Rules / Restrictions

- The `qconnector` operation stereotype is valid in process or facade classes only.
- Connector operations must have exactly one struct parameter.
- Connector operations may have a return type of void or a struct.
- The parameter and return structs may take any form, however the generated code is only capable of mapping structs which are “flat” - structs that do not aggregate other structs - and which have only fixed length fields. For complex structs, it is necessary to implement a mapper class to map the struct to and from messages. Examples of coding and decoding complex structs are provided below.

26.6 Encoding Methods for Fundamental Types

Datatype	Encoded Width	Encoding method
SVR_BLOB	Variable	Converted directly to a padded string
SVR_BOOLEAN	1	false = `0`, true = `1`
SVR_CHAR	1	Converted directly to a 1 character string
SVR_DATE	8	yyyyMMdd
SVR_DATETIME	15	yyyyMMddThhmmss (ISO 8601 standard)

Datatype	Encoded Width	Encoding method
SVR_DOUBLE	25	Numeric
SVR_FLOAT	16	Numeric
SVR_INT8	1	Numeric
SVR_INT16	6	Numeric
SVR_INT32	11	Numeric
SVR_INT64	21	Numeric
SVR_MONEY	25	Numeric
SVR_STRING	Variable	converted directly to a padded string
SVR_UNBOUNDED_STRING	N/A	Not natively supported

Table 26.1 Encoding methods

- `SVR_BLOB` and `SVR_STRING` are variable in that the length of the encoded message is equal to the length specified for that type in the model. If the data in the string is less than the maximum amount allowed, space padding is appended to the data in the message to bring it up to the maximum size.
- `SVR_UNBOUNDED_STRING` is not natively supported because the string length is not known at generate time and is required for creating fixed length messages. However it is possible for the developer to implement a custom mapper to handle unbounded strings.
- Numeric datatypes are converted to right-justified human-readable strings. For example: `45678`, `-23123`, `1000003.14159`, `1.4E-45`.

26.7 Using Customized Encoding/Decoding Classes

By default the encoding method used for each field in a struct used or returned by connector operations is based on the type of the field. For example, the mapper class for `curam.util.type.DateTime` is `curam.util.connectors.mqseries.MQFieldMapper.DateTimeMapper`; for boolean fields it is `curam.util.connectors.mqseries.MQFieldMapper.BooleanMapper`.

For any individual field in any operation it is possible to override this default and specify the name of the class which should be used to map the data. Names of the custom mapper classes are specified in the properties file `QueueConnectorFieldMappers.properties` which must be included in the application classpath.

Entries in the properties file take the following format:

`[class].[operation].[param].[field]=[mapper]`

where

- `[class]` is the name of the process or facade class containing the connector operation;
- `[operation]` is the name of the connector operation;
- `[param]` is the name of the parameter - or the property `return` to specify the return value for the operation;
- `[field]` is the name of the field within the parameter struct;
- `[mapper]` is the fully qualified class name of the required mapper class. This must be a subclass of `curam.util.connectors.mqseries.MQFieldMapper`.

```
MyBPO.connectorOp1.dtls.phoneNumber=com.acme.util.PNMapper
MyBPO.connectorOp1.return.phoneNumber=com.acme.util.PNMapper
```

Example 26.1 Sample QueueConnectorFieldMappers.properties

26.8 Example 1 - Working with Variable Length Fields

In the following example a custom field mapper class is used to implement a primitive variable length field message. The variable length field is encoded by prefixing the data with a six character string containing a number which specifies the length of the data in the remainder of the string.

Note that this example only shows the implementation at the Cúram end of the queue. The remote system will also need to understand the encoding method and implement the necessary translations using the language of choice on the remote system.

The following pseudo code describes the struct being used in the operation. Fields `idNumber` and `dateOfBirth` will use the default conversion methods for their type and will be converted into ten and eight character strings respectively. The `historyText` field is a variable length field and will be encoded and decoded by means of a custom mapper class.

```
struct PersonHistory {
  String<10> idNumber;
  String historyText;
  Date dateOfBirth;
}
```

Example 26.2 Pseudo code for the struct to be mapped:

Method `addToHistory` of class `LegacyBPO` sends a `PersonHistory` struct to a legacy system, the legacy system will append text to the variable length field `historyText` and return an updated copy of `PersonHis-`

tory.

```
interface LegacyBPO {
  PersonHistory addToHistory(dtls PersonHistory);
}
```

Example 26.3 Pseudo code for the BPO interface

Note that field `historyText` is being used in two cases - once in the parameter to operation `addToPersonHistory` and once in the return value from the operation. Therefore, the custom mapper class must be specified for each of these cases in `QueueConnectorFieldMappers.properties` (the lines are broken up for clarity).

```
LegacyBPO.addToPersonHistory.dtls.historyText=
  com.acme.mqutils.VariableStringMapper
LegacyBPO.addToPersonHistory.return.historyText=
  com.acme.mqutils.VariableStringMapper
```

Example 26.4 The property file entries linking the fields to the mapper

The following listing shows the implementation of the custom mapper class.

```
package com.acme.mqutils;
// implementation for variable length string field mapper class
public class VariableStringMapper
extends MQFieldMapper {

  /**
   * The size of a prefix at the beginning of the string
   * which specifies the length of following data.
   */
  private static final int kStringHeaderInfoLength = 6;

  /**
   * Gets the encoded version of the mapped field within
   * the given struct.
   *
   * @param object The struct class containing the
   *           mapped field.
   * @return The field encoded as a String.
   * @throws AppException if the field could not be encoded.
   *
   */
  public String encode(Object object)
  throws AppException {
    String historyText = null;
    // get the "historyText" field from the given struct:
    try {
      historyText = (String) getMappedField().get(object);
    } catch (IllegalAccessException e) {
      // use the handler in the superclass to deal with
      // this exception:
      handleEncodingException(e, object);
    }

    // construct the prefix which will hold the
    // size of the data.
    int bufferLength = historyText.length();

    String sizeSpecifierString = String.valueOf(bufferLength);
    // pad the size specifier to the right length
    sizeSpecifierString = MQUtils.padRight(
      sizeSpecifierString,
      kStringHeaderInfoLength);
```

```
    // put the prefix and the data together.
    String result = sizeSpecifierString + historyText;
    return result;
  }

  /**
   * Decodes the given string and assigns the resulting value
   * to the mapped field in the struct.
   *
   * @param object The struct class containing the field
   * @param encodedString The encoded form of the data.
   * @return the number of characters consumed from the
   *         encoded string.
   * @throws AppException if the target struct field could
   *         not be accessed.
   */
  public int decode(Object object, String encodedString)
  throws AppException {
    // the first N characters contain an expression
    // specifying the width of the encoded field.
    String sizeSpecifierString =
      encodedString.substring(0, kStringHeaderInfoLength);
    sizeSpecifierString = sizeSpecifierString.trim();
    int sizeOfString = Integer.valueOf(
      sizeSpecifierString).intValue();
    // Now that we know the size of the data, take that
    // many characters of data from the encoded string:
    String historyText = encodedString.substring(
      kStringHeaderInfoLength,
      kStringHeaderInfoLength + sizeOfString);
    // Update field "historyText" of the given struct:
    try {
      getMappedField().set(object, historyText);
    } catch (IllegalAccessException e) {
       // use the handler in the superclass to deal with
       // this exception:
      handleDecodingException(e, encodedString);
    }
    // indicate how many characters we decoded - remember
    // to include both the characters used to indicate the
    // string size AND the actual string data.
    return sizeOfString + kStringHeaderInfoLength;
  }
}
```

Example 26.5 Mapper class implementation for variable string

Examples of *MQSeries* messages transmitted and received by this connector operation are:

- `10000361iw4 One.19700714`
- `10000361iw9 One. Two.19700714`
- `10000361iw16 One. Two. Three.19700714`

Where the first 10 characters are the `idNumber` field, the last 8 characters are the `dateOfBirth` field and the middle section is the variable length `historyText` field, of which the first six characters specify the length of the data.

26.9 Example 2 - Working with Lists

In the following example a custom field mapper class is used to implement

encoding and decoding of a struct which aggregates a list of another struct. The list is encoded into a single string whereby the first 4 characters contain a number specifying the number of entries in the list and the remainder of the string consists of the encoded form of each struct as a fixed length string. The main purpose of this example is to illustrate how list aggregations are handled when implementing a custom mapper class.

As with the previous example, this example only shows the implementation at the Cúram end of the queue. The remote system will also need to understand the encoding method and implement the necessary translations using the language of choice on the remote system.

The following pseudo code describes the struct being used in the operation. Struct `PersonDtls` will be encoded as a fixed length 18 character string. Struct `PersonDtlsList` will be encoded by encoding each struct in its list, concatenating the results into a string, and prefixing the string with a six character string specifying the number of entries in the list.

```
struct PersonDtls {
  String<8> idNumber;
  String<10> surname;
}

struct PersonDtlsList {
  sequence <PersonDtls> dtls;
}
```

Example 26.6 Pseudo code for the structs to be mapped:

Method `processNames` of class `LegacyBPO` sends a `PersonDtlsList` struct to a legacy system, the legacy system will perform some processing on this data and return an updated copy of `PersonDtlsList`.

```
interface LegacyBPO {
  PersonDtlsList processNames(p1 PersonDtlsList);
}
```

Example 26.7 Pseudo code for the BPO interface

Again, as in the previous example, field `dtls` of struct `PersonDtlsList` is being used in two cases: once in the parameter to operation `processNames` and once in the return value from the operation. Therefore the custom mapper class must be specified for each of these cases in `QueueConnectorFieldMappers.properties` (the lines have been split for clarity).

```
LegacyBPO.processNames.p1.dtls=
  com.acme.mqutils.PersonDtlsListMapper
LegacyBPO.processNames.return.dtls=
  com.acme.mqutils.PersonDtlsListMapper
```

Example 26.8 The property file entry linking the fields to the mapper

The following listing shows the implementation of the custom mapper class.

```
package com.acme.mqutils;

// implementation
public class PersonDtlsListMapper {

  /**
   * The size of a prefix at the beginning of the string
   * which specifies the number of encoded entries in the
   * remainder of the string.
   */
  private static final int kStringHeaderInfoLength = 4;

  /**
   * The number of characters used to encode one
   * 'PersonDtls' struct.
   */
  private static final int kLengthOfOneEncodedStruct = 18;

  /**
   * Encodes the 'dtls' member into a string. The first 4
   * characters contain the number of items in the list, the
   * rest of the string consists of the encoded version of
   * each struct in the list concatenated together.
   *
   * @param object the object containing the field to be
   *         encoded
   * @throws AppException if it couldn't be encoded
   * @return A encoded string.
   */
  public String encode(Object object) throws AppException {

    PersonDtlsList.List_dtls d = null;

    try {
      // get a reference to the field within the struct
      // to be encoded
      d = (PersonDtlsList.List_dtls)
        getMappedField().get(object);
    } catch (IllegalAccessException e) {
       // use the handler in the superclass to deal with
       // this exception:
      handleEncodingException(e, object);
    }

    // construct the prefix which will specify the number
    // of items in the list.
    int bufferLength = d.size();
    String sizeSpecifierString =
      String.valueOf(bufferLength);

    // apply padding to make it the right size
    sizeSpecifierString =
      MQUtils.padRight(
        sizeSpecifierString, kStringHeaderInfoLength);

    // Now go through the items in the
    // list and encode each one.
    String data = "";
    for (int i = 0; i < d.size(); i++) {
      PersonDtls currentItem = d.item(i);
      data += encodeOneEntry(currentItem);
    }

    // put the prefix and the data together.
    String result = sizeSpecifierString + data;
    return result;
  }

  /**
   * Decodes a series of PersonDtls entries in the string
   * and adds them to field PersonDtlsList.List_dtls in the
   * given PersonDtlsList object.
```

```
  *
  * @param object The class containing the field to be decoded
  * @param encodedString the string containing the field data
  * @return a number indicating the number of characters decoded
  * @throws AppException if the string could not be decoded.
  */
public int decode(Object object, String encodedString)
throws AppException {

  PersonDtlsList.List_dtls dtls = null;

  // Get a reference to the list field within the object.
  // Note that we will be adding to this rather than
  // reassigning it.
  try {
    dtls = (PersonDtlsList.List_dtls)
      getMappedField().get(object);
  } catch (IllegalAccessException e) {
     // use the handler in the superclass to deal with
     // this exception:
    handleEncodingException(e, object);
  }

  // find out how many entries to be decoded.
  String header =
    encodedString.substring(0, kStringHeaderInfoLength);
  int numOfEntries =
    Integer.valueOf(header.trim()).intValue();

  // skip over the header.
  int chunkBegin = kStringHeaderInfoLength;

  // take chunks from the encoded string,
  // decode each one into an instance of the
  // struct, then add the struct to the list.
  for (int i = 0 ; i < numOfEntries; i++) {

    int chunkEnd = chunkBegin + kLengthOfOneEncodedStruct;
    String currentChunk =
      encodedString.substring(chunkBegin, chunkEnd);
    // encode one struct...
    PersonDtls newItem = decodeOneEntry(currentChunk);

    // and add it to the list:
    dtls.add(newItem);

    chunkBegin = chunkEnd;
  }

  // tell the caller the number of characters we consumed
  // from the encoded message.
  return chunkBegin;
}

/**
 * Encodes the struct into a string. Each field is padded
 * out to its maximum size, and the fields are concatenated
 * together to yield the result.
 *
 * @param d the struct to be encoded.
 * @return an encoded string containing the struct data.
 * @throws AppException If a field was too big to encode
 */
private String encodeOneEntry(PersonDtls d)
throws AppException {

  String result
    = MQUtils.padRight(d.idNumber, 8)
    + MQUtils.padRight(d.surname, 10);

  return result;
}
```

```
  /**
   * Decodes a string into an instance of the struct - does
   * the inverse of encodeOneEntry
   *
   * @param encodedEntry an encoded struct
   * @return a new instance of the struct
   * @see private String encodeOneEntry(PersonDtls)
   */
  private PersonDtls decodeOneEntry(String encodedEntry) {

    PersonDtls result = new PersonDtls();

    result.idNumber = encodedEntry.substring(0, 8).trim();
    result.surname = encodedEntry.substring(8, 18).trim();

    return result;
  }
}
```

Example 26.9 Mapper class implementation for list of structs

For example, the following list of `Ent18131` structs:

- `("0000361i", "James")`
- `("0024684x", "John")`
- `("8211519f", "Sharon")`

would be encoded as follows:

```
"3   0000361iJames
 0024684xJohn      8211519fSharon    "
```

where the first four characters contain a number specifying the number of encoded structs to follow, and the remaining string consists of three 18 character blocks corresponding to the three encoded structs.

Chapter 27

Subclassing

27.1 Introduction

The *IBM Cúram Social Program Management* SDEJ supports subclassing for `<<process>>`, `<<facade>>`, `<<entity>>`, `<<webservice>>`, and `<<wsinbound>>` classes and is intended to be used to add new functionality or override existing functionality. It cannot be used to add extra attributes to entities or structs.

27.2 Reasons for Subclassing

Reasons for using subclassing include:

- Adding new stereotyped methods to existing `<<entity>>` classes.
- Adding or contributing to an existing `<<entity>>`'s or operation's exit points.
- Modifying an existing `<<entity>>` operation's `Readmulti Max` options.

27.3 How to Model It

27.3.1 Basic Subclassing

A class is transformed into a subclass by adding a “generalization” relationship from the subclass to the superclass (base class). On a class diagram this appears as a line between the two classes with an arrow pointing toward the superclass.

This means that the subclass inherits all the operations of the superclass, and in addition it may:

- Add extra functions;
- Modify the applicable options of the function in the superclass.

Consider two classes where `MySubclass` is a subclass of `MyBaseClass`:

- `MyBaseClass` has two operations: `op1()` and `op2()`
- `MySubclass` has three operations: `op1()`, `op2()` and `op3()` where op1 and op2 is inherited from `MyBaseClass` and `MySubclass`.op3 is provided only in the `MySubclass` class.

27.3.2 Replacing the Superclass

When you define a subclass, you may specify that the subclass replaces its superclass entirely. To turn on the feature for an individual entity class the `Replace_Superclass` property in the *Rational Software Architect* Curam Properties tab must be set to "`1 - yes`" using the supplied drop-down.

For example, setting `Replace_Superclass` to `yes` for a class, `MySubclass`, means that instances of the base class, `MyBaseClass`, will no longer be created. All requests for the base class (`MyBaseClass`) will now receive an instance of the subclass (`MySubclass`). This is handled by the factory mechanism and is transparent to the user.

27.3.3 Abstract Classes

A class is made abstract by setting its `Abstract` option to `yes` in the meta-model. In this case the generated *Java* class hierarchy for this abstract class will not include a factory class. This means that the class cannot be instantiated and the only purpose of having the class is to enable it to be subclassed.

All non-abstract subclasses of the abstract classes will have the factory component and are instantiated in the normal way.

The developer must provide the `impl` *Java* code for abstract classes (unless the abstract class has no subclasses). From here on the usual rules for abstract classes apply: the `impl` class can contain implementations for some or all of the methods declared in the class, and any methods for which an implementation has not been provided must be implemented by the subclass(es).

27.3.4 Restrictions

- Multiple inheritance is not supported by the Cúram generators.
- Subclassing can only be used to add or override operations, it cannot be used to add or override attributes.

27.4 How to write Code for Subclassing

There are no specific restrictions on writing code for subclassing. It is possible to subclass any entity, facade or process class without having to change the way that class is declared or used.

New subclasses of existing classes should be written in new source files. All new source files should be placed within the source subdirectory of the `EJBServer\components\<custom>` directory. Where <custom> is any new directory created under the components directory that conforms to the same directory structure as `components\core`. The generated class hierarchy will dictate the packaging of the new source files.

27.5 Example - Using Subclassing to Override Entity Exit Points

27.5.1 Overriding Validation Exit Point

In order to override the validation exit point of an `<<entity>>` in a subclass:

- enable the `Automatic validation operation` option on an `<<entity>>` subclass;
- specify at least one of the `<<entity>>` superclass stereotype `<<insert>>` or `<<modify>>` operations in the subclass.

For example, consider two classes, `MyEntityClass` and `MyEntitySubClass`. The subclass, `MyEntitySubClass`, would inherit the `<<key>>` and `<<details>>` of the superclass. `MyEntitySubClass` would have the `Automatic validation operation` option enabled and would add the `<<insert>>` or `<<modify>>` operations.

For more information on validation exit point, see Section 6.9.3, *Validation*.

27.5.2 Overriding Pre Data Access, Post Data Access, and On-Fail exit points

In order to override the `Pre Data Access`, `Post Data Access`, or `On-fail` exit points of an `<<entity>>` in a subclass:

- specify the operation(s) of `<<entity>>` superclass in the subclass;
- enable the `Pre Data Access`, `Post Data Access`, or `On-fail` options as appropriate on the operations of the `<<entity>>` subclass.

For example, consider two classes, `EntityClass` and `EntitySub-`

Class with the subclass, EntitySubClass, inheriting the <<key>> and <<details>> of the superclass. The same operations would be defined in both classes; e.g.: <<insert>>, <<read>>, and <<modify>>. In both these classes these operations would have the following exit point options enabled:

- On Fail operation is enabled on operation insert;
- Post Data Access operation is enabled on operation read;
- Pre Data Access operation is enabled on operation modify.

For more information on exit points see Section 6.9, *Exit Points*.

Chapter 28

Application Customization

28.1 Overview

One of the more difficult aspects of customizing an application is the handling of upgrades to the original model at a later stage. Any changes which have been stored with the original model will be overwritten when a newer version of the model is taken on. This situation can be avoided by storing customizations separate from the original model. The original model can then be upgraded without overwriting any of the customizations.

It is also important to read the *Cúram Development Compliancy Guide* for more information about customizing the product.

The following features are available to facilitate the customization of an application:

- Chapter 7, *Extension Classes*;
- Section 5.3, *Overriding a Domain Definition*;
- Chapter 27, *Subclassing*.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typograph-

ical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products

should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 the names and addresses used by an actual business enterprise is entirely coincidental.

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 IBM, 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. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents intended programming interfaces that allow the customer to write programs to obtain the services of IBM Cúram Social Pogram Management.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/us/en/copytrade.shtml.

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

Oracle, Java and all Java-based trademarks and logos are registered trademarks of Oracle and/or its affiliates.

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