merative™

Cúram 8.1.3

Modeling Reference Guide

Note

Before using this information and the product it supports, read the information in Notices on page 117

Edition

This edition applies to Cúram 8.1, 8.1.1, 8.1.2, and 8.1.3.

Contents

1 Cúram modeling reference

The Cúram generator uses the UML meta-model to automatically generate all the required stubs, skeletons, classes, and communications that are required to interact with a database and remote clients. This approach allows the developer to concentrate on providing the application business logic.

1.1 Cúram modeling overview

Use the Cúram UML model to create client/server applications with a minimum of complexity. The UML model simplifies database access, EJB management, and client/server interaction.

The Cúram Generator uses the UML meta-model to automatically generate all the required stubs, skeletons, classes, and communications that are required to interact with a database and remote clients. By using the Cúram Generator, you can concentrate on the application business logic. The following guide describes the tools and components that you use to model the application, and how the Cúram Server Code Generator handles each component when it generates classes.

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

- Domains: application-specific data types. Domains are analogous to C++ typedefs.
- Entities: the objects that are modeled and persistently stored by the application. Entities correspond to relational database tables.
- Processes: related sets of activities to achieve a business goal.
- Structs: passed as messages throughout the application. Structs are analogous to structs in C+ +.
- Remote interfaces: client-visible interfaces that provide access to server functions.

You edit the UML meta-model with IBM®Rational® Software Architect. The Cúram Server Code Generator generates the code that produces the following outputs:

- JavaServer implementation code.
- Java beans.
- XML for the database entities and other classes in the model.

The Cúram Data Manager processes the XML produced by the Cúram Server Generator. The Cúram Data Manager produces the relevant SQL scripts that are used to create the required database structure for the application.

The following guide also provides a reference for Cúram model-based functions. Examples of these functions are Cúram domains, classes, operations, attributes and how they map to the underlying database.

1.2 UML overview

Use Rational® Software Architect Designer to create and maintain UML constructs. This model is referred to as the input meta-model.

The input meta-model provides the input for the Cúram Generator. It is a logical representation of the system. Use the meta-model mechanism to specify the content to generate.

The meta-model consists of a set of packages, which contain class representations. Classes have attributes and operations that have relationships with one another. Classes in the input meta-model result in various Java-generated classes and, in some cases, tables, and indexes in generated Database Definition Language (DDL). DDL is an SQL language subset that defines the structure and instances of a database in both a human-readable form and a machine-readable form.

UML and the input meta-model

UML constructs, which are created and maintained with Rational® Software Architect Designer, are collectively referred to as the input meta-model.

The meta-model provides input for the Cúram Generator. It is a logical representation of the system that you are developing. Use the meta-model mechanism to specify the content to generate.

The input meta-model consists of a set of packages, which in turn contain *class* representations, potentially containing *attributes* and *operations*, that have relationships with one another. Classes in the input meta-model result in various Java-generated classes and, in some cases, tables, and indices in generated Database Definition Language (DDL). DDL is an SQL language subset that defines the structure and instances of a database in both a human-readable form and a machine-readable form.

Architecture layers

The Cúram architecture is conceptually divided into layers: the remote interface layer, the business object layer, the data access layer, and the local interface layer.

Remote interface layer

The remote interface layer is an interface for business functions that a client program can use. The remote interface layer also interacts with third-party middleware components to ensure that the transactions that run in the business function are consistent and atomic.

Business object layer

The business object layer (BPO) implements the server's entire business functions.

BPOs represent the basic business entities that are modeled by the server application. BPOs implement the business logic of the Cúram server application. BPOs manipulate entity objects in a business-specific way. In business application development, concentrate most of your development effort in the BPO. For more information, see the *Process classes* related link.

Data access layer

The data access layer coordinates all interactions with the back-end database. For more information, see the *Entity classes* related link.

Local interface layer

Kubernetes The local interface layer provides interfaces for business functions that a client program can start, when client code is colocated within the same Java virtual machine (JVM) as the server code. The local interface layer is only supported on Cúram running on WebSphere® Application Server Liberty.

Related information

Stereotypes

Stereotypes are a Unified Modeling Language (UML) concept that further describes the various aspects of the Cúram application model. In UML, a stereotype is a string expression that assigns a classification to an object.

In general, stereotypes influence the behavior of the generator and determine its output. Therefore, for example, an entity class is identified by having a stereotype of entity and DDL and a data-access code that is produced by the generator.

The following lists the supported stereotypes.

Class stereotypes

Cúram class stereotypes include audit_mappings, extension classes, and facade classes.

The following table lists the class stereotypes, a short description of each, and a reference to a more detailed description.

Table 1: Cúram class stereotypes

Class stereotype	Description	Reference
audit_mappings	Use an audit mappings class to define extra fields in the database for auditing purposes.	1.4 AuditMappings classes overview on page 20
domain_definition	A domain definition is a meta-model class that defines a data type.	1.5 Domain definition classes on page 23
entity	An entity class encapsulates data-maintenance functions on a database table.	1.6 Entity classes on page 28
extension	Use extension classes to change the Audit Fields or Last Updated Field options of an entity or struct class.	1.7 Extension classes on page 38

Class stereotype	Description	Reference
facade	Use facade classes to create client-visible operations. Facade classes provide a simplified interface to a larger body of code, such as a class.	1.8 Facade classes on page 39
process	A process class encapsulates a business process.	1.9 Process classes on page 41
rest	Use a rest stereotype class for facades that contain nested lists which REST API services use.	
struct	A struct class is a meta-model representation of a Java class that contains a collection of fields.	1.10 Struct classes on page 43
wsinbound	A WS Inbound class represents an inbound web service.	*Cúram Web Services Guide*

Attribute stereotypes

Cúram attribute stereotypes include data items, defaults, and keys.

The following table lists the attribute stereotypes, a short description of each, and a reference to a more detailed description.

Table 2: Cúram attribute stereotypes

Stereotype	Description	Reference
audit_mappings	An audit field entry on the AuditMappings class.	1.4 AuditMappings classes overview on page 20
data item	An attribute on an RDO or ListRDO class.	Data item attribute
default	A public attribute or field in a struct class.	1.10 Struct classes on page 43
details	An attribute or field that is part of an entity but not part of the entity key.	1.6 Entity classes on page 28
key	An attribute or field that is part of an entity key.	1.6 Entity classes on page 28

Operation stereotypes

Cúram operation stereotypes include batch, nsinsert, and readmulti.

The following table lists the operation stereotypes, a short description of each, and a reference to a more detailed description.

Table 3: Cúram operation stereotypes

Stereotype	Description	Reference
batch	Process by the Batch Launcher by generated wrapper code.	batch on page 40 (facade class) and batch on page 42 (process class)
batchinsert	Use to insert large amounts of data by batch.	batchinsert on page 72
batchmodify	Use to modify large amounts of data by batch.	batchmodify on page 74
default	Use for standard non-database operation.	default on page 40 (facade class) and default on page 42 (process class)
insert	Use for a standard database insert.	Standard insert on page 61
modify	Use for a standard database update.	Standard modify operation on page 68
nkmodify	Use for a non-key database update.	Non-key modify on page 69
nkread	Use for a non-key database read.	Non-key read on page 66
nkreadmulti	Use for a non-key database read.	Non-key readmulti on page 67
nkremove	Use for a non-key database delete.	Non-key remove on page 72
ns	Use for a database operation for handcrafted SQL.	Non-standard on page 75
nsinsert	Use for a non-standard database insert.	Non-standard insert (generated SQL) on page 62
nsmodify	Use for a non-standard database update.	Non-standard modify (generated SQL) on page 69
nsmulti	Use for a database operation for handcrafted SQL.	Non-standard multi on page 76
nsread	Use for a non-standard database read.	Non-standard read (generated SQL) on page 65
nsreadmulti	Use for a non-standard database read.	Non-standard readmulti (generated SQL) on page 65
nsremove	Use for a non-standard database delete.	Non-standard remove (generated SQL) on page 71
qconnector	Use for connecting to external JMS.	qconnector on page 40 (facade class) & qconnector on page 43 (process class)
read	Use for a standard database read.	Standard read on page 63

Stereotype	Description	Reference
readmulti	Use for a standard database read.	Standard readmulti on page 64
remove	Use for a standard database delete.	Standard remove on page 70
wmdpactivity	Use for deferred processing.	wmdpactivity on page 40 (facade class) and wmdpactivity on page 42 (process class)

Relationship stereotypes

Cúram relationship stereotypes include assignable, extension, and foreignkey.

The following table lists the attribute stereotypes, a short description of each, and a reference to a more detailed description.

Table 4: Cúram relationship stereotypes

Stereotype	Description	Reference
aggregation	Use an aggregation to embed or nest instances of one type of class within another type of class.	1.21 Aggregation on page 84
assignable	Use an assignable relationship to map differing or exclude fields for an assign function.	1.22 Assignable on page 88
extension	The link between an extension class and a target class.	1.7 Extension classes on page 38
foreignkey	A modeled description of a database foreign key.	1.23 Foreign keys on page 92
index	A modeled description of a database index.	1.24 Indexes on page 94
uniqueindex	A modeled description of a database unique index.	1.25 Unique indexes on page 95

Data types

The input meta-model supports a number of data types that provide abstraction for you from the different underlying data types that the database, middleware, and Java layers use. Use these data types to define attributes, arguments, and return values in a way that is platform-neutral and database-neutral.

Use the SDEJ to map the attributes, arguments, and return values to the appropriate data type in each layer of the application.

The following table lists the Cúram data types and a short description of each.

Table 5: Cúram data types

Type	Description
SVR_BLOB	Use SVR_BLOB to hold binary data. Corresponds to class `curam.util.type.Blob`. Requires a size qualifier that is used only if the field is used on a database table. Fields of type SVR_BLOB might be null on the database.
SVR_BOOLEAN	Use SVR_BOOLEAN to holding binary values. Corresponds to the primitive Java type `boolean`. 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	Use SVR_CHAR to hold single character values. Therefore, you cannot use this data type to hold strings or arrays of characters. SVR_CHAR does not take a size qualifier. Corresponds to the primitive Java type `char`. Fields of type SVR_CHAR cannot be null on the database.
SVR_DATE	Use SVR_DATE to hold 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	Use SVR_DATETIME to hold 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	Use SVR_DOUBLE to hold floating point numbers. Corresponds to the primitive Java type `double`. Fields of type SVR_DOUBLE cannot be null on the database.
SVR_FLOAT	Use SVR_FLOAT to hold floating point numbers. Use to hold floating point numbers. Corresponds to the primitive Java type `float`. Fields of type SVR_FLOAT cannot be null on the database.
SVR_INT8	SVR_INT8 is an 8-bit integer. Corresponds to the primitive Java type `byte`. Fields of type SVR_INT8 cannot be null on the database.
SVR_INT16	SVR_INT16 is a 16-bit integer. Corresponds to the primitive Java type `short`. Fields of type SVR_INT16 cannot be null on the database.

Type	Description
SVR_INT32	SVR_INT32 is a 32-bit integer. Corresponds to the primitive Java type `int`. Fields of type SVR_INT32 cannot be null on the database.
SVR_INT64	SVR_INT64 is a 64-bit integer. Corresponds to the primitive Java type `long`. Fields of type SVR_INT64 can be null on the database.
SVR_MONEY	SVR_MONEY is a fixed-point numeric value with two decimal places that hold 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	Use SVR_STRING to hold string values. Corresponds to the Java class `java.lang.String`. A SVR_STRING can, optionally, have a length qualifier. A SVR_STRING without a length qualifier is a SVR_UNBOUNDED_STRING. Strings that are stored on the database must have a length qualifier to enable a maximum size to be specified for the database column. You can store a SVR_STRING on the database as either CHAR, VARCHAR or CLOB, depending on its size and the type of database. Fields of type SVR_STRING can be null on the database.
SVR_UNBOUNDED_STRING	Use SVR_UNBOUNDED to hold string values for which it is not necessary to specify a maximum length. Corresponds to the Java class `java.lang.String`. SVR_UNBOUNDED_STRING is the only Cúram data type that you cannot use by an attribute of an entity class. You cannot use the SVR_UNBOUNDED_STRING data type to specify its maximum size or to define a database column. To define a string field on an entity, you must use SVR_STRING with a length qualifier.

1.3 Packages

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 structure of the hierarchy is significant because options that are set at the package level apply to all classes and other packages within that package. Any option can be overridden in any of the subpackages by setting the option at that level to a different value.

The CODE_PACKAGE option

The CODE_PACKAGE option, when specified, affects struct, entity, facade, and process classes within that package and in the packages contained within that package.

Two or more process or struct classes in the model can have the same name. Equally named classes are distinguished (on the server side only) by their CODE_PACKAGE value, which might be specified for one of its containing packages.

As noted, the CODE_PACKAGE option, when specified, affects struct, entity, facade, and process classes within that package and in the packages contained within that package. When you apply the CODE_PACKAGE option to a class, it moves that class into a package within the default package, `curam`, and includes any of the package's parent CODE_PACKAGE options. The following example outlines how the CODE_PACKAGE option works.

The UML meta-model class `MyProcess` in the model creates the following Java classes:

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

You must implement:

- `<ProjectPackage>.impl.MyProcess`

If you want to create another class that is named `MyProcess`, you can do so if you create the class within a package for which a different CODE_PACKAGE option was specified. Creating a `MyProcess` class ensures that you can store the corresponding Java classes in separate locations on disk.

You specify the following option for the package that contains the `MyProcess` class (the option must be manually typed into the documentation for the package in the UML meta-model):

- CODE_PACKAGE = `custom`

The following classes and interfaces result:

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

You must implement `<ProjectPackage>.custom.impl.MyProcess`.

Rules for the CODE_PACKAGE feature

You must apply specific rules to the CODE_PACKAGE feature.

Apply the following rules to the CODE_PACKAGE feature:

- CODE_PACKAGE values must be valid Java identifiers.
- Setting the CODE_PACKAGE option for a package recursively affects subpackages and process, facade, entity, and struct classes within the package.
- If you specify a CODE_PACKAGE value in a package whose parent also specified a CODE_PACKAGE value, the value that you specify overrides the value that is specified by the parent.

  The following is an example where the value that you specify overrides the value that is specified by the parent:

  - 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 because a literal blank is ignored by the generator, and cannot be used to override a non-blank setting.
- You can specify multiple level code packages by using 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, but only one instance of each facade class name can exist. Cúram clients currently cannot distinguish between multiple facade classes with the same name, regardless of their CODE_PACKAGE setting.
- Like process and struct classes, the behavior of the CODE_PACKAGE option with entity classes results in generated interface and struct classes that are produced in different packages. However, entity class names must still be unique throughout the application regardless of the CODE_PACKAGE option setting. The reason is 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. This code package might not be the same code package as the operation that produced them.

1.4 AuditMappings classes overview

Audit fields contain extra information about the modification history of each record. You can add audit fields to database tables for auditing purposes.

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

You set the information in the audit fields. You must include the following information:

- Creation time
- Modification time
- Program ID
- User ID

Audit fields consist of all the attributes of a special class in the input meta-model called `AuditMappings`. You can automatically add a field corresponding to each attribute of this special class to the database table and, also, to all the standard details structs for the entity.

AuditMappings classes rules

Specific rules apply to the `AuditMappings` class and to the `AuditMappings` implementation class.

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”, that is, the class cannot aggregate any other classes.

You can make an audit mapping available to application by adding a class that is 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 provide a Java implementation class for it in the `impl` package.

Note: If this implementation class is not present, the server application cannot be compiled. To address this, perform ONE of the following actions:

- Delete the `AuditMappings` class from the model.

OR

- Explicitly disable audit mappings completely by specifying the generator switch -noauditmappings option.

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

- The class must contain the same fields as defined in the meta-model, that is, they must have the same name and data type.
- The fields must be public.
- The class does not need to inherit from any other class.
- Optionally, the class can contain the `public void set(final boolean isInsert, final boolean isModify)` method.

  The `public void set(final boolean isInsert, final boolean isModify)` call-back method that is called whenever necessary (such as during inserts and modifies) by the data access layer. Do not use this call back method to populate the fields of the `AuditMappings` class. The two boolean parameters indicate whether the database operation is an insert or a modify.
- Optionally, the class can contain a public void method that is called `set`, which takes no parameters. The data-access layer calls this method whenever fields require updating. Any public methods whose names start with `set` and take no parameters are called in arbitrary order. Support for using multiple setter methods will be discontinued.

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

The corresponding fields of the audit mapping fields that are not present in the details struct are updated on the database. Therefore, it is unnecessary to include the audit mapping fields in the details struct in order to update the audit mapping fields on the database. However, the audit mapping fields are not included in table-level auditing.

AuditMappings classes outputs

Turning on auditing for an entity affects fields and the infrastructure data-access code. Use specific operation stereotypes to set the audit information.

The following are the effects of turning on auditing for an entity:

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

Use the following operation stereotypes to set audit information:

- `modify`
- `nsmodify`
- `insert`
- `nsinsert`
- `nkmodify`
- `batchinsert`
- `batchmodify`

AuditMappings classes options

Two options are available for attributes of the `AuditMappings` class in the model. Depending on the option that you set, specific operations do not change the value of the applicable audit mapping field.

The following two options are available for attributes of the `AuditMappings` class in the model:

- Exclude from modify
- Exclude from insert

If the Exclude from modify option is set for an audit mappings field, then the value of this field is not changed by any of the following operations:

- `modify`
- `nsmodify`
- `nkmodify`

Therefore, the field is set when a record is inserted and is never changed by subsequent updates. Similarly, if the Exclude from insert option is set, then the value of the field is not set by any of the following operations:

- `insert`
- `nsinsert`

Instead, the value of the field is changed by any subsequent updates. The default value for each of these options is `false`.

Note: It is not possible to exclude audit mapping fields from operations of the `ns` stereotype. Use handcrafted SQL in these operations to access audit mapping fields directly.

Note: If your audit mappings include a time stamp, then populate this field with the value that is returned by `TransactionInfo`, `getProgramTimeStamp()`. This value ensures that all audit mapping-enabled tables that are modified during the transaction have the same time stamp value even though they were not written to at the same time.

1.5 Domain definition classes

Domains are data type definitions that resolve to a primitive data type or another domain.

In relational database terminology, a domain defines the permitted range of values for an attribute of an entity. In Cúram, domain definitions work in a similar way. Equivalent primitive types are supported across client, middleware, server, and database components of a Cúram application.

Table 6: Domain primitive types at different levels 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

Working with domains, rather than primitive types, means that you are not required to manage different representations of data in the various application layers. For this reason, you must define entity and structure attributes in terms of a previously defined domain. You cannot use primitive data types directly.

You can also define validations on each domain type in the client application. You can then execute a specific validation for all attributes that are defined in terms of a domain type before transactions are invoked on the server. This client-side, pre-flight validation provides the user with feedback on basic data type validation without the need to call the server. The subsequent reduction in failed transactions results in lower network traffic.

Domain definition class options

15 domain definition options are available.

Code table names

A code table name contains the valid entries for the domain definition. If the domain definition represents a hierarchy of code tables, use the name of the lowest code table in the hierarchy to specify the code table name.

Editable fields

For fields where a code table is specified, the client application displays a drop-down list of valid values for the field if the field is editable. For a code table hierarchy where 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.

Read-only fields

For fields where a code table is specified, the client application displays a code table translation for the field if the field is read-only. For a code table hierarchy where the code table is read-only, only the translation for the lowest level code table is displayed.

This option is only valid for domain definitions that you defined in terms of one that has the Code Table Root option set to `yes`.

Code table roots

A code table root specifies whether the current domain definition is the root of a hierarchy of a code table domain definition.

If the code table root is set to `yes`, then all domain definitions that are defined in terms of it must specify the Code Table Name option. If you forget to specify the Code Table Name option, the generator displays an error.

As this domain definition holds code table codes, the domain definition type must match that of a Cúram code table code, that is, SVR_STRING<10>.

Related reference

Code table names on page 24
A code table name contains the valid entries for the domain definition. If the domain definition represents a hierarchy of code tables, use the name of the lowest code table in the hierarchy to specify the code table name.

Compress embedded spaces

A compress embedded space specifies that any extra whitespace, rather than all whitespace, is embedded in the string and that all leading and trailing whitespace is removed before it is sent to the server.

A whitespace character consists of any character for which java.lang.Character.isWhitespace(char) returns true. These characters include the space character, the tab character, and the line-feed character.

Extra whitespace consists of a run of whitespace characters immediately *after* another whitespace character. This method 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 that are separated by three spaces are converted to the pair of words that are separated by one space.

In cases where the first whitespace character is not a space, the results might not be as expected. For example, for a pair of words that are separated by a carriage-return, a line-feed, and a space, space is converted to the pair of words that are separated by the carriage-return character.

If you use this feature on multiple-line text fields, it removes indentation.

Note: Switching on this option also trims leading and trailing whitespace from the string, regardless of the Remove Leading Spaces and Remove Trailing Spaces option settings.

Convert to uppercase

Convert to uppercase converts the contents of the string field to uppercase before it sends the string field to the server.

Custom validation function name

Use the custom validation type to specify the name of the function that associates the custom validation type with the application UML model.

Domain definition validations that are implemented in the client infrastructure include a custom validation type that corresponds to a developer-supplied function. The function performs validations on data users who are entered via the client interface.

Use this option to specify the name of this function that associates it with the application UML model. The value of the option must be the name of a function (that is, just the function, not class + function as the class name is defaulted in the client code). It must also be a valid Java identifier.

Note: This feature is deprecated. For more information on the new domain plug-in system, see the *Custom Data Conversion and Sorting* related link.

Related reference

Default option
A default option specifies that the default option field contains a default value after it is displayed.

Maximum size
The maximum size specifies the maximum number of characters that you can enter in the field before the field is sent to the server. The maximum size is the field storage size on the database. The feature is implemented in the Cúram client application and the database DDL.

Maximum value
The maximum value specifies the maximum numeric value that you can enter in the field before the field is sent to the server.

Minimum size
The minimum size specifies the minimum number of characters that you must enter in the field before the field is sent to the server.

Minimum value
The minimum value specifies the minimum numeric value that you can enter in the field before the field is sent to the server.

Multibyte expansion factor
For string domains, the multibyte expansion factor specifies an expansion factor, float from 1.0 to 4.0, to apply when multibyte character set MBCS data is used with Db2 or DB2 for z/OS.

The multibyte expansion factor overrides the global build-time property `curam.db.multibyte.expansion.default.factor`. Use only the multibyte expansion factor to deviate from the global setting, for example a specific domain is exceeding a Db2 limit. A setting of 1.0 effectively turns off expansion for this domain. You can set this option for domains in instances where you know that the contents never contain localized data. For example, the domains are constrained to programmatically defined Western characters and they cannot be input by using a client.

The same option set for a string entity attribute can override this domain setting. For more information, see the *Multibyte expansion factor* related link. This option is ignored if the feature is turned off (the `curam.db.multibyte.expansion` property set to `false` at build time). For more information, see the *Planning for MBCS Data* related link.

Related reference

Multibyte expansion factor on page 46
The multibyte expansion factor is an override for the domain-level multibyte expansion factor. The multibyte expansion factor only applies to string entity attributes.

Pattern match
A pattern match specifies a regular expression that the string value must match before it is 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 that is used in Java 1.5. For more information on the supported syntax for these regular expressions, see the JavaDoc documentation for the `java.util.regex.Pattern` class that is supplied with your Java SDK.

Remove leading spaces

The remove leading spaces option specifies that any leading spaces are removed from the string before the string is sent to the server.

Remove trailing spaces

The remove trailing spaces option specifies that any trailing spaces are removed from the string before the string is sent to the server.

Storage type

The storage type option specifies the type of string storage data type to use for the domain definition on the database. The storage type applies only to string domain definitions where a length is specified.

Overriding a domain definition

Use the Server Development Environment (SDEJ) to override existing domain definitions without modifying the original domain definition. Use this feature in situations where the original domain definition is provided by a third party. Therefore, do not modify this feature locally.

The following are the suggested uses for overriding a domain definition:

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

Domain definition override examples

To override a domain definition, create a new domain definition with the same name that is prefixed by an asterisk.

For example, to override the domain definition PERSON_NAME create a domain definition named *PERSON_NAME. At build time, the system uses the overridden version, complete with its own data type and options, instead of the original version.

Domain definition override considerations and limitations

Overriding a domain definition affects all usages of the original domain definition.

It is your responsibility to ensure that pre-existing functionality is not broken by overriding domain definitions. Attempting to change the type of the domain definition, the code table name, or the code table root is not recommended.

Domain definition usage rules

You can override a domain definition only once.

The following rules apply to using domain definitions:

- You cannot override a domain definition override. For example, if *PERSON_NAME overrides PERSON_NAME you *cannot* further override *PERSON_NAME with **PERSON_NAME.
- You cannot create overrides for domain definitions that do not exist. For example, if there is a domain definition override named *PERSON_NAME then the model must contain a domain that is named PERSON_NAME.

- You cannot use domain definition overrides as attributes of structs or entities, that is, attributes cannot use domain definitions where the name of the domain definition begins with an asterisk.

1.6 Entity classes

An entity is a collection of fields and associated database operations. Entity classes are the fundamental building blocks of systems that are developed with Cúram. They correspond to database tables. The Cúram generator supports automatic code generation for entity classes.

Entity classes have a stereotype of entity. 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 read, insert, modify, remove, readmulti (read multi reads multiple records from a table based on a partial key).

Standard operations such as read or insert operate on a single database table by default.

Entities can have attributes, operations, dependencies, inherits relations, and aggregations. A set of rules is associated with each of these constructs.

Entity class rules

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

Entity 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 because Cúram interface objects are stateless and atomic. Attributes are contained within generated standard key and details structs.

The stereotype of an entity attribute cannot be blank. It must be either a details attribute or a key attribute. For information about struct class outputs, see the *Struct class outputs* related link.

Related reference

Struct class outputs on page 44
Use input meta-model struct classes to 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.

Details attribute

The details attribute is included as a column on the database table and in the standard details struct for the entity.

For more information about standard details structs, see the *Standard details structs* related link.

Related reference

Standard details structs on page 30

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

Key attribute

The key attribute is included as a column on the database table.

The key attribute forms part of the primary key. The key attribute is included in both the standard details struct and the standard key struct for the entity. For more information about standard key structs, see the *Standard key structs* related link.

Related reference

Standard key structs on page 30
Standard key structs are generated for entity classes. Standard key structs contain those attributes in the class whose stereotype is key. If no such attributes exist in the class, then no standard key struct is generated.

Entity operations

Entity operations can be divided into two categories as determined by their stereotype: database and non-database operations.

Database operations

Database operations are operations where the generator recognizes the stereotype.

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

Non-database operations

Non-database operations are operations where the generator does not recognize the stereotype.

For non-database operations, the generator only creates prototypes and skeletons. It does not generate data-access operations. You must implement in the BOL the body of these functions.

The operations available for entity classes are listed in operation stereotypes. For more information, see the *Operation stereotypes* related link.

Related reference

Operation stereotypes on page 14
Cúram operation stereotypes include batch, nsinsert, and readmulti.

Entity 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.

Standard key structs

Standard key structs are generated for entity classes. Standard key structs contain those attributes in the class whose stereotype is key. If no such attributes exist in the class, then no standard key struct is generated.

Use the standard key struct as a parameter for operations that require a primary key. For example, read and delete operations.

Standard key structs do not appear in the input meta-model, but you can use them as arguments to operations in the input meta-model. When you are assigning a name to standard key structs, use the name of the corresponding entity with the word `Key` appended. For example, the standard key struct for the class `Employer` is `EmployerKey`.

Standard details structs

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

Standard details structs do not appear in the input meta-model, but you use them as arguments to operations in the input meta-model.

When you are assigning a name to standard details structs, use the name of the corresponding entity with the word `Dtls` appended. For example, the standard details struct for the class `Employer` is `EmployerDtls`.

Standard list structs

Standard list structs are generated for entity classes that contain one or more operations of stereotype readmulti or nkreadmulti. This struct contains a single attribute, `dtls`, that 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` is `EmployerDtlsList`.

Entity class options

The options available for entity classes are entity class abstracts, allow optimistic locking, audit fields, enable validation, last updated field, No Generated SQL, and replace superclass.

Entity class abstracts

The entity class abstract specifies that the class is abstract. Abstract classes are typically subclassed by other classes.

For more information about abstract classes and subclassing, see the *Subclassing* related link. .

Related reference

Subclass modeling on page 113

Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

Allow optimistic locking

Optimistic locking only applies to entities that are not subclasses.

Optimistic locking is supported on certain database operations. To use optimistic locking on an entity's operation, you must first switch on this option for the class.

Related reference

Operations on page 47
Operations represent the functions of modeled classes that can, depending on their type, be handcrafted or generated by the Cúram generator.

Optimistic locking for concurrency control on page 32
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.

Audit fields

Audit fields only apply to entities that are not subclasses.

You can configure extra fields to store additional information on a database table for auditing purposes. For more information on these fields, see the *AuditMappings classes overview* related link.

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

Related reference

AuditMappings classes overview on page 20
Audit fields contain extra information about the modification history of each record. You can add audit fields to database tables for auditing purposes.

Enable validation

The validation operation is an exit point that automatically calls to validate data. This exit point is called before the data-access layer entity operations whose stereotype is `insert` or `modify`.

For more information about exit points, see the *Exit points* related link.

Related reference

Exit points on page 34
An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

Last updated field

The last updated field is only applicable to entity classes that are not subclasses.

To use the last updated field feature for an entity class, first switch on the feature. The feature adds an extra timestamp field to the specified entity. Typically, the extra timestamp field is updated with the current date and time whenever the record is written. However, an exception is where the write was performed by an `ns` operation. For more information on the last updated field, see the *Last updated field* related link.

Related reference

Last updated field on page 37

The last updated field is a field that you can add to database tables to contain extra information about the modification time of each record for reporting purposes.

No Generated SQL

No Generated SQL switches on the No Generated SQL for all database operations of the entity class.

Use individual entity operations to override the value of the No Generated SQL option.

For more information, see the *No Generated SQL* related link.

Related reference

No Generated SQL on page 53
Use No Generated SQL to avoid generating data access code. As a result, you can provide your own implementation.

Replace superclass

The replace superclass option is only relevant to entities that are subclasses.

If you set the replace superclass option, then requests to create instances of the superclass create the subclass. Use this feature to change functionality by replacing subclasses with other classes.

Optimistic locking for concurrency control

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 modifies the record, another user cannot modify it without first rereading 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 the user is editing is locked for update only while the changes are being committed. Locking the record in this way 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, it is not guaranteed that the user will succeed. An update that relies on optimistic locking fails if another user updates 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 that the record is modified. The generated DAL code checks this version number while the record is being updated. 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 it is written to the database.
- `nsinsert`- If optimistic locking is enabled on an entity class, you must include the version number field in the details struct. The details struct is automatically initialized before it is written to the database.

Optimistic locking is only possible for operations that modify a single database record and whose details struct includes the generated version number field. Therefore, for non-standard operations, you must 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 must be VERSION_NO. You must ensure that the model contains a numeric domain definition named VERSION_NO.

To support optimistic locking on an operation, you must do two things:

- Enable the Allow Optimistic Locking option on the entity.

  By enabling Allow Optimistic Locking, the version number field is automatically added to the entity.
- Enable the Optimistic Locking option on the operation.

  By enabling Optimistic Locking, the generator generates code in the DAL for the operation checks and updates the record version numbers accordingly.

Table-level auditing

Use the Database table-level auditing option to enable table-level auditing.

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

The information that is 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. When you switch on this option, the generated data-access code records audit information for an operation.

The type of audit information that is 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.

Information captured by table-level auditing

Table-level auditing captures information such as the date and time of the transaction and the ID of the user who invoked the transaction. Before and after audit information about the record is only captured if optimistic locking is switched on.

The following information is captured by table-level auditing:

- *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 modified table.

- *Program name* - The FID of the invoked transaction.
- *Transaction type* - Indicates whether the transaction was *online*/*batch*/*deferred*/ and so on.
- *Operation type* - Indicates whether the operation was of one of the following: create, read, update, or delete.
- *Key info* - The key that is provided to this operation. This key can identify one or many records.
- *Details of changed data* - Logged details of the changed data in an XML format.
  You can see the exact format of this XML in the JavaDoc details for the class `curam.util.audit.AuditLogInterface` in the *`doc/api`* directory of the SDEJ. The JavaDoc includes the names of the fields that are 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 it is guaranteed that only a single record was affected by the operation. Therefore, the audit information includes information about the record before and after the operation. The old version of the record is reread. The old value of each field is compared to the new value, and any changed field is included in the audit information, that is, unchanged fields are filtered out.

  If optimistic locking is switched off, then, for performance reasons, the record is not reread during the update. Therefore, the audit information contains only the new versions of all the fields that are involved in the update, and 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 that case, you cannot record a *before-after* comparison of the update. All the detail fields are included, regardless of whether the new value is different to the old value.

  You can compress this data when you use the default auditing handler by specifying the `curam.audit.audittrail.datacompressionthreshold` property. For more information about this property, see the *Cúram Configuration Parameters* related link.

Related concepts

Audit information storage

By default, the captured audit information is written to the AuditTrail database table.

You can also supply your own auditing handler by specifying a class that implements the `curam.util.audit.AuditLogInterface` interface. For more information, see the *Customization* related link.

Related reference

Exit points

An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

Four types of exit points are supported: pre-data access, post-data access, validation, and on-fail.

Pre-data access

The pre-data access 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`, for example, `preread`.

Post-data access

The post-data access function is called after the DAL function.

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

Validation

The validation function is called before standard insert and standard update operations, and also before the pre-data access functions. It provides a common place to put validation code.

The validation 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 only insert and modify are guaranteed to pass in the standard details struct, it is only these operation stereotypes that 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.

On-fail

The on-fail function is called if an error occurs in the data access function.

The on-fail function is named after the method to which it belongs, prefixed with `onFail`, for example, `onFailread`.

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

Exit point parameters

In most cases, the parameters to an exit point method must include the parameters to the method to which the exit point belongs and the return type of the method to which the exit point belongs.

For more information about validate exit points, see the *Validation* related link. Except for validate exit points, the parameters to an exit point method consist of the following:

- The parameters to the method to which the exit point belongs. (If you specify any extra parameters for a database operation in the model, this is the only place that you 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 is not included in the parameters of exit point methods for nsread and ns operations.

Use the following approach to generate the return type parameter into the parameters of exit point methods for nsread and ns operations:

- Add an unstereotyped method to the entity class to give 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 then calls the nsread or ns operation, and can access its return value as required.

- For on-fail exit points, an exception class. The exception class is the exception that is thrown from the Data Access Layer. The exit point can handle the error or pass it on by throwing it.

Related reference

Validation on page 35

The validation function is called before standard insert and standard update operations, and also before the pre-data access functions. It provides a common place to put validation code.

Exit point uses

Use exit points for validation or for completing a business process.

For example, after you modify an invoice detail-line record, use the last modified date to update the invoice header record.

Exit point misuses

Do not use exit points to populate incomplete fields in incoming parameters.

To populate incomplete fields of incoming parameters, wrap the database function in a non-database function that performs the following:

- Copies the incomplete record.
- Completes the missing fields.
- Invokes the database operation.

When you add an exit point to an entity operation, ensure that the exit point does not affect other users of the operation.

Entity inheritance

Input meta-model entity classes can subclass other entity classes.

Typically, entity classes are subclassed to add functionality (for example, stereotyped operation) required for special processing of the associated database table but that do not belong in the parent class.

For more information about subclassing, see the *Subclassing* related link.

Related reference

Subclass modeling on page 113

Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

Entity inheritance usage rules

Four entity inheritance rules apply. The entity rules relate to entity classes and subclasses.

The following rules apply when you use entity inheritance:

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

Last updated field

The last updated field is a field that you can add to database tables to contain extra information about the modification time of each record for reporting purposes.

The last updated field feature is similar to the `AuditMappings` feature.

The last updated field 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 *lastWritten* field of type SVR_DATETIME is automatically added to the entity.
- Unless the write was performed by an `ns` operation, the Cúram infrastructure automatically populates this field with the current time whenever the record is written to the database.

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

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

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

  Figure 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 result in the *lastWritten* field to be set:

- `insert`

- `nsinsert`
- `modify`
- `nkmodify`
- `nsmodify`
- `batchinsert`
- `batchmodify`

Note: Unlike the version number field that the optimistic locking feature uses, 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 is enabled for an entity, the field is always updated for the preceding operation stereotypes by the infrastructure data access code. Regardless of whether the field is present in the structure you are using, the field is always updated for the preceding operation stereotypes by the infrastructure data access code.

1.7 Extension classes

Use extension classes to specify options for a target class without modifying the meta-model definition of the target class.

Each extension class must link to one target class. At build time, the contents of an extension class are effectively super-imposed on its target class.

Extension classes usage

To extend an existing class, create a new class of stereotype extension.

For more information about using and modeling with Rational® Software Architect Designer, see the *Working with the Cúram Model in Rational Software Architect Designer* related link.

You add options to the extension class in the same way as for other classes. When you add any of these options to an extension class, they add (if not existing) or modify (if existing) the same named option on the target class.

When you create an extension class in Rational® Software Architect Designer, ensure that the settings for the extension class are compatible with the class you are extending. The reason for this is that when you create an extension class it can apply to different class types.

You link the extension class to its target class by adding a relationship of stereotype extension between the two classes. Create the new class within a custom subpackage of your model.

Related concepts

When to use extension classes

Restrict your use of extensions classes to two specific purposes.

The following are the only circumstances in which to use extension classes:

- To switch on the Last Updated Field option on an entity.

- To switch on the Audit Fields option on an entity or struct.

Note: Rational® Software Architect Designer allows you to specify additional information such as attributes and options for extension classes but only include the preceding two options in the class. Other changes are not compliant.

Extension classes: considerations and limitations

In the relationship between two classes in Rational® Software Architect Designer, the relationship is stored as a free-standing object in a package. The relationship is not stored within either of the actual classes.

Usually, the relationship between two classes is stored in the package that contains the diagram on which it was drawn. However, the relationships between the two classes are not always stored in the package. Ensure that you store the relationship 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 the inheritance relationships.

Extension classes: usage rules

Ensure that you apply three specific rules when you use extension classes.

The following rules apply to using extension classes:

- You can apply an extension class to only one target class.
- You can extend a class by multiple extension classes.
- You can apply extensions to classes of stereotype entity, struct.

1.8 Facade classes

Facade classes encapsulate a business process that is visible to the client. It is a collection of operations. Facades are the business object layer of the application.

Facade classes do not have data maintenance operations or any relationship with database tables. Instead, facade classes manipulate other entity and process classes in order to implement a business process. Facade classes have a stereotype of facade.

Facade class rules

Ensure that you apply five specific rules when you use facade classes.

The following rules apply to using facade classes:

- 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; facade classes cannot inherit from entity or process classes.
- Facade classes cannot have attributes.

- Facade classes cannot have the same name.

Facade classes operations

Four operations are supported within facade classes: default, batch, wmdpactivity, and qconnector.

default

The default stereotype provides a standard or plain operation.

batch

For operations of stereotype batch, the Cúram generator produces the necessary source code wrappers to build a batch wrapper program. This program then enables the Batch Launcher to run the operation.

For more information about batch operations, see the *Developing batch processes* related link.

The following rules apply to defining batch operations:

- 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.

Related concepts

wmdpactivity

You can designate a method of a facade class as a deferred processing method by setting its stereotype to wmdpactivity.

For more information, see the *Cúram Server Developer* related link.

Related information

qconnector

For operations of stereotype qconnector, the generator produces the necessary source code to connect to a JMS provider (for example, IBM MQSeries).

For more information about JMS, see the *Cúram JMS queue connectors* related link.

Related reference

Cúram JMS queue connectors on page 100
Use Cúram connectors in a Cúram application to connect to other systems through JMS queues.

Facade class options

Three options are available for facade classes: abstract, generate facade bean, and replace superclass.

Abstract

The abstract specifies that the class is abstract.

Abstract classes are intended to be subclassed by other classes. For more information about abstract classes and subclassing, see the *Subclass modelling* related link.

Related reference

Subclass modeling on page 113
Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

Generate facade bean

The generate facade bean creates a stateless session bean for the facade class.

Use the generate facade bean class to enable your server to be accessed by other systems or by message-driven beans.

Replace superclass

Use replace superclass if the applicable facade class was subclassed from another class.

For more information about subclassing, see the *Subclass modelling* related link.

Related reference

Subclass modeling on page 113
Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

1.9 Process classes

Process classes are a collection of operations that encapsulate a business process.

Process classes do not have data maintenance operations, or any relationship with database tables. Process classes manipulate other entity and process classes to implement a business process.

For example, a banking system has an account transfer process which debits money from one account and credits another. Internally, the process uses an account entity to debit one account and credit another account. The process class itself is not responsible for any database manipulation. Instead, the process class packages a sequence of entity operations to execute the modeled business process. Process classes have a stereotype of process.

Business Process Objects

Business Process Objects (BPOs) are classes that reside in the Business Object Layer (BOL) of a Cúram server application, that is, the architectural layer between the Remote Interface Layer (RIL) and the Data Access Layer (DAL).

All business logic is implemented in the BOL. Consequently, BPOs constitute most of the handcrafted coding that is required to create a server application.

BPOs do not *directly* communicate with the RDBMS (which is implemented, largely automatically, in the DAL) or the middleware (which is implemented, largely automatically, in the BOL). Instead, BPOs specifically implement business logic.

Process class rules

Five specific rules apply to process classes.

Use the following rules for process class:

- 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; process class cannot inherit from entity classes.
- Process classes cannot have attributes.
- Two or more process classes can contain the same name only on the condition that different CODE_PACKAGE values are specified for each name.

Process class operations

Four operations within process classes are supported: default, batch, wmdpactivity, and qconnector.

default

The default stereotype offers a standard or plain operation.

batch

For operations of stereotype batch, the Cúram generator produces the necessary source code wrappers to build a batch wrapper program that enables the Batch Launcher to run this operation.

For more information about the Batch Launcher, see the *Developing batch processes* related link.

The following rules apply to defining batch operations:

- 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.

Related concepts

wmdpactivity

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

For more information, see the *Cúram Server Developer* related link.

Related information

qconnector

For operations of stereotype qconnector, the generator produces the necessary source code to connect to a JMS provider (for example, MQSeries).

For more information about JMS, see the *Cúram JMS queue connectors* related link.

Related reference

Cúram JMS queue connectors on page 100
Use Cúram connectors in a Cúram application to connect to other systems through JMS queues.

Process class options

Three process class options apply: abstract, Generate Function Identifiers (FIDs), and replace superclass.

Abstract

Abstract classes specify that the class is abstract.

Use abstract classes as a subclass of other classes. For more information about abstract classes and subclassing, see the *Subclass modelling* related link.

Related reference

Subclass modeling on page 113
Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

Generate Function Identifiers (FIDs)

The Generate Function Identifiers (FIDs) class specifies that a FID must be generated for it.

Replace superclass

Replace superclass applies only if this process class was subclassed by another class.

For more information about subclassing, see the *Subclass modelling* related link.

Related reference

Subclass modeling on page 113
Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

1.10 Struct classes

Struct classes are Java classes with public attributes and no modeled methods. A struct is the Java equivalent of a C++ struct. Use structs to group domain definitions and other struct classes to form programmatic record definitions.

Use struct classes as arguments to operations of entity and process classes. Use structs to package arguments to avoid long argument lists. Struct classes can also aggregate each other. These aggregations become struct members.

For example, a bank account entity has parameters to a read operation that consists of a key struct and a details struct. The key struct contains a single field for the account number. The details struct contains several fields, that is, Name, Balance, and so on. Struct classes have a stereotype of struct.

Struct class rules

Specific rules apply to struct classes.

Apply the following eight rules to struct classes:

- Struct classes must have one or more attribute or aggregation, that is, 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 can aggregate entity classes or other struct classes.
- Struct classes cannot be part of inheritance relationships.
- Struct classes that are used as key or details parameters to non-standard database operations must not aggregate other structs, that is, they must be "flat".
- Two or more struct classes can contain the same name once different CODE_PACKAGE values are specified for each, that is, similarly named struct classes must be distinguishable by having CODE_PACKAGE settings.
- In most cases, you must define a struct to use it as a parameter to an operation. The exception to this rule is standard key and details structs. Standard key and details structs are generated automatically by the Cúram generator are available for you to use.

Struct class outputs

Use input meta-model struct classes to 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. For example, zero for numerics, empty string for Strings, and so on. As each field is initialized to its default value, you do not need to use null values.

Each field is accompanied by comments that describe the domain definition hierarchy for the datatype.

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

Struct classes have no counterpart in generated DDL.

Struct class options

You can enable the audit fields struct class option.

Audit fields

Enabling audit fields automatically adds the available pre-configured audit fields to this struct.

Enable audit fields to use the struct class as a write operation of an entity where Audit Fields are already enabled .

For more information about audit fields, see the *AuditMappings classes overview* related link.

Related reference

AuditMappings classes overview on page 20
Audit fields contain extra information about the modification history of each record. You can add audit fields to database tables for auditing purposes.

1.11 Attributes

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 determines how the Cúram generator processes the UML meta-model.

Attribute rules

Class stereotypes map to attribute stereotypes. Four rules apply to attribute stereotypes.

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

Table 7: Mapping of class and 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
wsinbound	N/A

The following rules apply to 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 require.

Attribute options

The following attribute options are available: allow NULLs and multibyte expansion factor.

Allow NULLs

The allow NULLs option is only available for the details stereotyped attribute on an entity class.

The allow NULLs option determines whether NULL values are permitted on the corresponding database field. Setting this option to `no` means that a Not Null qualifier is included with this field in the generated DDL script.

The default value for this option depends 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`.

For more information about NULLs, see the *Null considerations* related link.

Related reference

[“Null” considerations on page 81]
When you write a handcrafted SQL statement, some Cúram datatypes are stored as null on the database if they are empty (that is, in their initial state).

Multibyte expansion factor

The multibyte expansion factor is an override for the domain-level multibyte expansion factor. The multibyte expansion factor only applies to string entity attributes.

For more information, see the *Multibyte expansion factor* related link.

The multibyte expansion factor specifies that an expansion factor (float from 1.0 to 4.0) is applied when multi-byte character set (MBCS) data is used with DB2 or DB2 for z/OS. It operates with its equivalent domain option and the global build-time properties `curam.db.multibyte.expansion.default.factor` and `curam.db.multibyte.expansion`. For more information about these properties, see the *Cúram Server Developer Guide* related link.

The multibyte expansion factor option is only necessary for DB2 MBCS data in order to deviate from the global or domain settings. For example, you might set this option to 1.0 (which effectively turns off expansion) for a string attribute where you know that the contents never contain localized data. An example is where the contents are constrained to programmatically defined Western characters and cannot be input via a client. This option is ignored if the feature is turned off by using the property `curam.db.multibyte.expansion`.

For more information about deviating from the global or domain settings, see the *Planning for MBCS Data* related link.

Related concepts

Related reference

Multibyte expansion factor on page 26

For string domains, the multibyte expansion factor specifies an expansion factor, float from 1.0 to 4.0, to apply when multibyte character set MBCS data is used with Db2 or DB2 for z/OS.

Related information

1.12 Operations

Operations represent the functions of modeled classes that can, depending on their type, be handcrafted or generated by the Cúram generator.

The following rules apply to operations:

- Operations must belong to either entity, process, facade, or wsinbound classes.
- Operations can be fully handcrafted or can use the features that are offered by standard operations.
- Operations cannot be individually hidden or exposed to clients; only whole classes can be hidden or exposed.

Operation rules

Six rules apply to using structs for operations.

The following rules apply 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, can be domain definitions.

Operation options

A total of 24 operation options are available.

Audit Business Interface (BI) calls to this operation

The Audit Business Interface specifies whether Business Interface-level auditing is performed for this operation.

Business Interface-level auditing applies to client-visible operations only. 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, and so on).

You can override this option at application startup-time by using application properties. For more information, see the *Database table-level auditing* related link.

Related reference

Database table-level auditing on page 50
Table-level auditing records provide information about the changes that are made to actual data on the database table. Use database table-level auditing to specify whether to perform table-level auditing for this operation.

Auto ID field

The Auto ID field specifies the field to use as the Auto ID field.

The Auto ID field only applies for certain insert operations of entity classes. During the insert, the Auto ID field is automatically populated with a generated unique ID to ensure that the record is uniquely identified.

Auto ID key

Use the Auto ID key to specify the key set from which a unique ID is generated.

Use the Auto ID key option only in conjunction with the Auto ID Field option.

Business Date

Use the Business Date option to specify that one field of the operation parameters is treated as the Business Date Field for the operation. As a result, the value of this parameter to the operation becomes the Business Date during the transaction.

The Business Date option applies to operations of a process class only. The Business Date is the `Date` or `DateTime` that is returned by the following methods:

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

The main purpose of the Business Date option is to give greater flexibility when you run batch programs where processing dates are significant. This applies, for 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()`. Depending on the day on which it is run, this program processes a different set of records.

Consider what would happen if you needed to regenerate 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 is run. Note that you need to ensure that this doesn't affect other users, so no other users can use the machine while you are changing the system date.

- Ensure that your batch request is the only one in the queue.
- Run the batch launcher to prompt 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 includes a Business Date field you only need to perform the following steps:

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

Syntax for the Business Date option

The Business Date option uses one of two formats.

Specify the Business Date option in one of the following formats:

- *fieldName*
- *paramName*. *fieldName*

where

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

- 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 either to `arg1.effectiveDate` or because it is the first (and only) parameter: `effectiveDate`.

- 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 the Business Date option

The Business Date option only applies to operations that correspond to individual server transactions.

These operations are the operations of facade classes and batch operations. Note that it does *not* apply to workflow activity or deferred processing operations.

The following rules apply to the Business Date option:

- The field that 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 that are invoked directly from Java code because this action does not result in starting a new server transaction.

- 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 is not overridden for that transaction. In this case, the Business Date for the transaction will either be the current system date or the overridden value that is specified in application properties. For more information, see the `Date` and `DateTime` JavaDoc documentation.

BytesMessage encoding character set

The BytesMessage encoding character set only applies to connector operations of process classes.

For more information, see the *qconnector operations options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

Database table-level auditing

Table-level auditing records provide information about the changes that are made to actual data on the database table. Use database table-level auditing to specify whether to perform table-level auditing for this operation.

Database table-level auditing only applies to database operations of entity classes.

The behavior of auditing depends on whether optimistic locking is switched on or off for the operation. For more information about auditing, see the *Table-level auditing* related link.

You can override the option at application startup-time by using application properties. This functionality is available to Audit BI. This option and the following is an example of how to use it.

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.

Use the two options to target 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 capture data to table AuditTrail.

Properties whose names begin with `curam.audit.opaudittrail` apply to the Audit BI calls option and capture data to table OpAuditTrail.

Example (1): To switch on the 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 the 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.opaudittrail.curam.core.facade.Participant.modifyAddress
```

to `false`.

Note:

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

Related reference

Table-level auditing on page 33
Use the Database table-level auditing option to enable table-level auditing.

Field Level Security

Apply Field Level Security to the fields returned by client-visible operations (that is, operations of the facade class). It only applies to operations of a client-visible operation (defined in a facade class).

In Rational® Software Architect Designer, use the **Secure Fields** properties tab of the facade class operation 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 operation, use the **Secure Fields** button from the properties tab for the operation. Click the **SID Name** cell for the returned **Field Name** to enter the SID. The maximum length of a security identifier is 100 characters.

The client infrastructure ensures that fields for which a SID is specified can only be viewed by users to whom that SID is granted. Fields for which no SID is specified are visible to all users.

The information about Field Level Security, which SID is assigned to a field, is written by the generator to an XML file. The Data Manager then loads the XML file into the database table FieldLevelSecurity. You must change the Data Manager configuration file *datamanager_config.xml* to reference the generated file *<ProjectName>_FieldsReturned.xml*. To change the Data Manager, add an entry to the **initial** target as shown in Figure 1.

```
<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>
```

Figure 2: Sample datamanager_config.xml for adding field level security information to the database

Once you add the field names and SIDs to the FieldLevelSecurity table, the SIDs are loaded into the SecurityIdentifier to assign them to groups. To assign them to groups, use the database command in Figure 2.

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

Figure 3: Inserting field level security SIDs into the infrastructure SecurityIdentifier table

Use the Security Administration console to assign these SIDs to user groups.

JNDI name of the QueueConnectionFactory class

The JNDI name of the QueueConnectionFactory class only applies to qconnector operations of process classes.

For more information about qconnector operations, see the *qconnector operation options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

JNDI name of the transmission queue

The JNDI name of the transmission queue only applies to qconnector operations of process classes.

For more information about qconnector operations, see the *qconnector operation options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

JNDI name of the reply queue

The JNDI name of the reply queue.

For more information about qconnector operations, see the *qconnector operation options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

Message type

The message type only applies to qconnector operations of process classes.

For more information about qconnector operations, see the *qconnector operation options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

No Generated SQL

Use No Generated SQL to avoid generating data access code. As a result, you can provide your own implementation.

The No Generated SQL option only applies to database operations of entity class.

The following is an example of using No Generated SQL. If the No Generated SQL is set to `yes` for a standard `read` operation that is named `myRead`, the generator produces a declaration of an abstract method that is named `myRead_da` with the same signature as the formerly generated `myRead` method. You must provide the implementation of method `myRead_da` as shown in Figure 1.

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

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

Figure 4: Handcrafted data access implementation for a standard read

For `readmulti` operations, that is 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*. You implement readmulti operations in Cúram by using the visitor design pattern whereby a subclass of `curam.util.dataaccess.ReadmultiOperation` is passed into the data access operation that then invokes its `operation(Object)` for each record that it finds. Usually, this operation adds the record to a collection that is returned to the caller.

For more information, see the *Cúram Server Developer* related link.

Crucially in `readmulti` operations, you return data 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. For an example, see Figure 2.

```
/*
   * 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;
  }
```

Figure 5: Handcrafted data access implementation for a readmulti

Related information

On Fail operation

Use the On Fail operation option to enable the on-fail exit point.

The On Fail operation option only applies to database operations of entity classes.

If any error occurs in the Data Access Layer (DAL), the On Fail operation is invoked. A copy of the parameters is 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 that occurred. The error can either be handled in this exit point or the exception can be thrown from here so that the error is managed elsewhere.

For more information, see the *Exit points* related link.

Related reference

Exit points on page 34
An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

Optimistic locking

Use optimistic locking to enable optimistic locking for this operation.

Optimistic locking only applies to certain update operations of entity classes.

You can only use optimistic locking if the Allow Optimistic Locking option is set for the entity class.

For more information on optimistic locking, see the *Optimistic locking for concurrency control* related link.

Related reference

Optimistic locking for concurrency control on page 32
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.

Order by

Use the order by operation to specify the fields by which a sequence of records are sorted as they are read from the database.

The order by operation only applies to entity operations of the following stereotype: `readmulti`, `nsreadmulti`, and `nsreadmulti`.

For the order by operation, any or all of the fields of an entity are valid arguments. Records are always sorted in ascending order.

If you do not specify the order by option, records are returned in arbitrary order.

Post-data access operation

Use the post-data access operation to determine whether a standard database operation has a post-exit point.

The post-data access only applies to database operations of entity classes.

For more information on exit points, see *Exit points* related link.

Related reference

Exit points on page 34
An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

Pre-data access operation

Use the pre-data access operation to determine whether a database operation has a pre-exit point.

The pre-data access operation only applies to database operations of entity classes.

For more information on exit points, see *Exit points* related link.

Related reference

Exit points on page 34
An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

Readmulti_Max

Use the Readmulti_Max operation to specify the maximum number of records that are returned by a readmulti operation.

The Readmulti_Max option only applies to entity operations of the following stereotype: `readmulti`, `nsreadmulti`, `nkreadmulti`, and `nsmulti`.

If there are more records available than the Readmulti_Max, then handling is based on the setting of the Readmulti_Informational option. Unless the Readmulti_Informational setting is on in the model for the operation, there is no Readmulti_Max enforcement.

If the Readmulti_Informational option is not specified, then Readmulti_Max uses the generator system default.

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

For more information, see the *Readmulti_Informational* related link.

Related reference

Readmulti_Informational on page 56
Use the Readmulti_Informational operation to determine the handling of the system when the specified Readmulti_Max is reached.

Readmulti_Informational

Use the Readmulti_Informational operation to determine the handling of the system when the specified Readmulti_Max is reached.

The Readmulti_Informational option only applies to entity operations of the following stereotype: `readmulti`, `nsreadmulti`, `nkreadmulti`, and `nsmulti`.

By default, a Readmulti_Max message is logged and it returns all entries to the user. If this option is specified, then you can add an InformationalMessage to the current transactions InformationalManager for handling in the Application's Facade layer. In this case, only the specified Readmulti_Max number of entries is returned to the user.

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

For more information, see the *Readmulti_Max* related link.

Related reference

Readmulti_Max on page 56
Use the Readmulti_Max operation to specify the maximum number of records that are returned by a readmulti operation.

Response message timeout (seconds)

The response message timeout (seconds) operation only applies to qconnector operations of process classes.

For more information, see the *qconnector operation options* related link.

Related reference

qconnector operation options on page 101
Use the appropriate qconnector operation.

Security

Use the security operation to determine whether to apply security to the operation.

The security option only applies to client-visible records.

If you enable security for an operation, the generator produces code in the Remote Interface Layer (RIL). The RIL then checks whether the user is authorized to invoke the operation. If the user is not authorized to invoke the operation, an exception is thrown.

SQL

Use the SQL option to supply the SQL code that is executed by the operation. The generator converts the supplied SQL into DAL (Java and SQL) code.

This option only applies to the entity operations of the following stereotype: `ns` and `nsmulti`.

For more information, see the *Using handcrafted SQL in non-standard entity operations overview* related link.

Related reference

Using handcrafted SQL in non-standard entity operations overview on page 81

Transactional

Use the transactional operation to specify whether a transaction is started for an operation.

The transactional option only applies to client-visible operations.

Where

Use the where operation to specify a custom WHERE clause for the generated SQL that the DAL code uses for this operation.

The where option only applies to operations of the following entity classes: `readmulti` and `nsreadmulti`.

1.13 Operation parameter options: mandatory fields

Mandatory fields are fields that you must populate when displayed on a client page. Use the mandatory fields operation to specify mandatory fields for any parameter.

You must populate the mandatory option value with a single-line, comma-delimited string.

Figure 1 is an example of an operation signature.

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

Figure 6: Operation Signature

The pseudo-code for the structures that are involved as parameters in this operation is outlined in Figure 2.

```
// 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;
}
```

Figure 7: Pseudo-Code for Parameter Structures

To make the following fields mandatory for the operation parameter, perform these steps:

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 you add mandatory fields that are contained in structures aggregated by the parameter type class, the mandatory fields must be fully qualified by the relevant aggregation role names as shown.

1.14 Entity operations overview

A database operation is an operation of an entity class whose stereotype is recognized by the Cúram generator. The generator produces data-access Java code that is based on the stereotype for these operations.

The generator treats all other operations as if their stereotype was blank and produces Java interfaces and factories for them. The generator does not produce data-access code for these other operations. Instead, you must provide your own implementation for these operations.

Standard operations

The following standard operations are available: standard single-record and standard multi-record.

Standard single-record operations

Standard single-record operations are the most basic type of operation.

For standard single-record operations, the database only returns a single row and it is not required to model any arguments. The reason for this is that the code generator assumes standard key and details structs where appropriate.

The standard single-record operations are represented by the following operation stereotypes:

- `insert`
- `modify`
- `read`
- `remove`

Standard multi-record operations

Use standard multi-record operations to process multiple rows (instead of operating on a single database table row).

In database maintenance applications, it is often necessary to return multiple records to a user interface. The user then selection one record for processing. Batch programs also frequently operate on multiple rows of a table. For example, a printing batch program for bank account statements typically operates on the accounts of every client on record.

Standard multi-record operations are represented by the operation stereotype `readmulti`.

Non-standard operations

The following non-standard operations are available: generated SQL and handcrafted SQL operations.

Generated SQL operations

Non-standard generated SQL operations are similar to the standard operations, except that it is not assumed that the arguments and return type are standard key and standard details structs.

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. You must 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.

As you define the key struct of a non-standard generated SQL operation, you can define a key struct that does not uniquely identify a single record. As a result, certain operations might not behave as expected. For example, for a non-standard modify operation all records that match the key are modified (not just the intended record).

The generated SQL operations are represented by the following operation stereotypes:

- `nsinsert`
- `nsmodify`
- `nsread`
- `nsreadmulti`
- `nsremove`

Handcrafted SQL operations

Non-standard handcrafted SQL operations are the most flexible type of operation that is provided by the generator.

Use non-standard handcrafted SQL operations to specify custom parameters and SQL for the operation. The only parameters that are generated for `ns` operations are the ones that you provide. All parameters that you provide are replicated into all the generated layers of the application.

Use this type of operation for situations where none of the other operations are suitable. These situations include joins across tables and queries that count or calculate max, and so on.

Non-standard handcrafted SQL operations are represented by the following operation stereotypes:

- `ns`
- `nsmulti`

Non-key operations

Non-key operations operate on all rows of a database table. Typically, you use non-key operations on tables that contain one row.

Non-key operations are represented by the following operation stereotypes:

- `nkmodify`
- `nkread`

- `nkreadmulti`
- `nkremove`

Batch operations

Batch programs use operations that are tailored specifically to the batch environment.

For more information about batch programs, see the *Developing batch processes* related link.

Batch operations are represented by the following operation stereotypes:

- `batchinsert`
- `batchmodify`

Related information

1.15 Entity insert operations

Use insert operations to insert, or add, a row onto a database table.

Insert operations operate on a single row at a time. The following are the two types of insert operation:

- `insert`
- `nsinsert`

Standard insert

A standard insert operation contains a stereotype insert.

Standard insert description

Standard insert operations insert a single record onto the appropriate database table by using the information that is passed in a standard details struct.

You do not need to specify arguments for standard insert operations in the input model. You can specify extra arguments and exit points that can access these arguments for the operation. The extra arguments and exit points do not affect the generated code.

Standard insert use

Use a standard insert operation to create a new record on a database table and to update each attribute.

You do not need to specify any arguments for the standard insert operation in the input model. Generated standard key and details structs are assumed as arguments, where appropriate.

You can specify extra arguments and exit point that can access these arguments for the operation. The extra arguments and exit points do not affect the generated code.

You can also use this pattern with the generation pattern for the Auto ID Field sequence number.

Standard insert parameter and generator notes

Standard insert operations use the entity's details structure as an input parameter.

The input parameter is automatically generated and contains all the fields of the record.

- *Parameters* (None)
- *Return value* (None)
- *Generator action* (The generator adds the standard details struct as a parameter)

Non-standard insert (generated SQL)

A non-standard insert operation uses a stereotype `nsinsert`.

Non-standard insert description

Using the information that you provide from a non-standard details struct, non-standard insert operations insert a single record onto the database table of the parent entity.

Non-standard insert use

Use a *non-standard insert* operation to create a new record on a database table where you do not need to update each attribute.

Attributes that are 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. The application designer must decide whether the improved efficiency is worth the extra complexity of more operations on your entities.

Use a non-standard insert where you know that the database can perform the operation significantly more efficiently or where the operation is required by a high-volume transaction.

Non-standard insert operations take a single input parameter, that is, a structure that defines the attributes to insert. Each attribute of this structure must match some entity attribute by name and type.

Non-standard parameter and generator notes

A warning is displayed if a non-standard operation has a non-standard details parameter that does not include fields that cannot be null.

For more information, see the *Null considerations* related link.

Fields that are not included in the details struct are not initialized, that is, the DBMS sets them to <null>.

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

Related reference

“Null” considerations on page 81
When you write a handcrafted SQL statement, some Cúram datatypes are stored as null on the database if they are empty (that is, in their initial state).

1.16 Entity read operations

Depending on the type of operation and arguments you provide, read operations return one or more rows from a database table.

The following are entity read operation types:

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

Standard read

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

Standard read description

Standard read operations read a single record from a database table into a standard details struct. Standard read operations use a standard generated key struct (that is, the primary key) as their search criteria.

You do not need to specify arguments for standard insert operations in the input model. You can specify extra arguments and exit points that can access these arguments for the operation. The extra arguments and exit points do not affect the generated code.

Standard read use

Use a standard singleton read operation to read all the attributes of a specific database record.

Standard singleton read operations use the primary key of an entity to locate the target record. You can only create standard singleton read operations for entities with primary keys. As the primary key of an entity is unique, a standard singleton read always returns a single database record.

Standard read parameter and generator notes

Standard singleton read operations use the entity's key and details structures as input and output parameters respectively.

The parameters are automatically generated and are not specified in the UML meta-model.

You are not required to specify arguments for these operations in the input model. Generated standard key and details structs are assumed as arguments where appropriate.

You can specify extra arguments and exit points that can access these arguments for the operation. The extra arguments and exit points do not affect the generated code.

- *Parameters* (None.)
- *Return value* (None.)

- *Generator action* (The generator adds the standard key struct as a parameter and the standard details struct as the return value.)

Standard readmulti

A standard readmulti operation has a stereotype readmulti.

Standard readmulti description

Standard multiple read operations use an input parameter that you designate as the key structure for the operation.

The return value is a structure that contains a list of the entity's details structures. You must specify the first parameter. However, as the return value is automatically generated, the first parameter is not specified in the UML meta-model.

Standard readmulti use

Use a standard multiple read operation to read all the attributes of a set of database records. This operation is based on a key that you specify.

The stipulation about efficiency of keyed access, as described for non-standard read, modify, and remove operations, also applies to multiple reads. As a result, the designer must ensure the efficient use of database indices.

Standard readmulti parameter and generator notes

A standard readmulti operation takes a partial key struct and returns a list of standard details structs. Every record that matches the criteria is then returned in the list.

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

To change the arbitrary order in which records are returned, use 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 a record or records 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 creates a list wrapper for the standard details struct for the entity and adds it as the return value for the operation.)

Non-standard read (generated SQL)

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

Non-standard read description

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

Non-standard read use

Use a *non-standard read* operation either to read a subset of the attributes on a database record or 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. At development, there is no guarantee that only one record is targeted. If there is more than one record in the result set, a runtime error is generated.

Non-standard read operations can be more efficient than standard read operations. The reason for this is that non-standard read operations result in less database I/O.

As with any operation where you specify the key, there is no guarantee that the database can access the target records efficiently. Instead, the designer must define appropriate indices to ensure this efficiency.

Non-standard parameter and generator notes

Non-standard read operations use key and details structures (as input and return types, respectively) that you must create 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. You can also use standard (generated) key or details structures.

- *Parameters* (A non-standard key struct to specify the record to retrieve. The key must have the capacity to uniquely identify a single record. If more than one record matches the criteria, an exception is thrown.)
- *Return Value* (A non-standard details struct that contains the retrieved data.)
- *Generator action* (None.)

Non-standard readmulti (generated SQL)

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

Non-standard readmulti description

Non-standard readmulti operations take a partial key struct and a details struct as input meta-model parameters.

Non-standard readmulti operation return a list of the provided details struct. Every record that matches 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 while it is assumed that a standard readmulti is the standard generated details struct for the entity. For non-standard readmulti, you must 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.

Non-standard readmulti use

Use a non-standard multiple read operation to read a subset of the attributes of a set of database records, based on a key that you specify.

The designer must ensure the efficient use of database indices when it is reading the indices based on this key.

Non-standard readmulti parameter and generator notes

Like standard operations, non-standard multiple read operations use an input parameter that you designate as the key structure for the operation.

The return value that you specify in the UML meta-model is a structure that contains the attributes that you want to return for each record that is 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 that contains the list is automatically generated).

- *Parameters* (A non-standard key struct to specify the search criteria for the record or records 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 the attributes to return from the readmulti operation. The members of this struct must be a subset of the fields of the entity.)
- *Generator action* (The generator creates a list wrapper for the non-standard details struct that you specify, and uses the list wrapper as the return value for the operation.)

Non-key read

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

Non-key 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. Non-key operations execute SQL statements that do not have a where clause, that is, they operate on all rows on a table.

For a non-key read operation, you must have a single row on the table. Typically, you use this type of operation to read a value from a control table that contains a single record.

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

Non-key use

Use a non-key singleton read operation to read a record from a database table where there is a single record.

If the database table contains more than one record, the non-key operation generates a runtime error.

Typically, you use the non-key operation for control tables that contain a single record.

Non-key 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, the operation throws an exception.

- *Parameters* (None.)
- *Return value* (None.)
- *Generator* (The generator adds the standard details struct as the return value.)

Non-key readmulti

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

Non-key readmulti description

Non-key readmultis are similar to standard readmulti operations. The only difference is that the non-key readmultis return all rows of a table rather than those that match a partial key.

Non-key operations, as the name suggests, do not take a key parameter. Non-key readmultis execute SQL statements that do not have a where clause, that is, they operate on all rows on a table.

For a non-key read operation, there must be a single row on the table. Typically, you use this type of operation to read a value from a control table that contains a single record.

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

Non-key readmulti use

Use a non-key multiple read operation to read all the attributes of the records on a database table.

Non-key readmulti parameter and generator notes

Non-key multiple read operations do not take a key argument. The value the non-key multiple read operation returns is a structure that contains a list of the entity's details structures as an output parameter.

You do not specify 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, that is, a list of standard details structs for the entity.

- *Parameters* (None.)
- *Return value* (None.)
- *Generator action* (The generator creates a list wrapper for the standard details struct for this entity and uses the list wrapper as the return value.)

1.17 Entity update operations

Depending on the type of operation and arguments you provide, the entity update operation modifies data in one or more rows of the database table.

The following are types of entity update operation:

- `modify`
- `nsmodify`
- `nkmodify`

Standard modify operation

A standard modify operation has a `modify` stereotype.

Standard modify operation description

Use a standard modify operation to modify a specific record on a database table.

You do not need to specify any arguments for the standard modify operation in the input model. Use a generated standard key struct to specify the record to modify. The modified data is contained in a generated standard details struct. You can specify extra arguments. You can access these arguments by exit points for the operation. They have no effect on the generated code.

Standard modify operation use

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

You can also use the standard modify pattern with the optimistic locking pattern.

Standard modify parameter and generator notes

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

- *Parameters* (None.)
- *Return value* (None.)
- *Generator action* (The generator adds the standard key struct and standard details struct as parameters.)

Non-standard modify (generated SQL)

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

Non-standard modify description

Use the non-standard modify operation to update records on the database table of the parent entity with information from a non-standard details struct that you provide.

Non-standard modify use

Use a *non-standard modify* operation 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. A possible result is that multiple records are updated. You also specify the attributes of the entity to update.

Non-standard modify operations can be more efficient than standard modify operations because non-standard modify operations involve less database I/O. Also, the database might not need to update as many indices relative to using a standard modify operation.

Non-standard parameter and generator notes

Non-standard modify operations use non-standard key and details structures as input parameters. You must create the input parameters 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. You can also use standard (generated) key or details structures.

- *Parameters* (A non-standard key struct to specify the record to modify. The non-standard key can specify multiple records. In this case, all records that match the non-standard key are updated.)

  A non-standard details struct that contains the updated version of the data.
- *Return value* (None.)
- *Generator action* (None.)

Non-key modify

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

Non-key modify 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 that do not have a where clause, that is, they operate on all rows on a table.

For a non-key read operation, there must be a single row on the table. Typically, you use this type of operation to read a value from a control table that contains a single record.

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

Non-key modify use

Use a *non-key modify* operation to update all the records on a database table.

The attribute values of each record are set to the values that you specify in the parameter to the non-key modify function.

Typically, only use a non-key modify operation for control tables that contain only one record.

Non-key parameter and generator notes

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

- *Parameters* (None.)
- *Return value* (None.)
- *Generator action* (The generator adds the standard details struct as a parameter.)

1.18 Entity delete operations

Depending on the type of operation and arguments you provide, delete operations remove one or more rows from the database table.

The following are entity delete operation types:

- `remove`
- `nsremove`
- `nkremove`

Standard remove

A standard remove operation has a stereotype `remove`.

Standard remove description

Standard remove operations delete a specific record from a database table.

You do not need to specify any arguments for standard remove operations in the input model. Use a generated standard key struct to specify the record to delete. You can specify extra arguments. Exit points for the operation can access these arguments. The extra arguments do not affect the generated code.

Standard remove use

Use a standard remove operation 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. As the primary key of an entity is unique, a standard remove always deletes a single database record.

Standard remove parameter and generator notes

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

- *Parameters* (None.)

- *Return value* (None.)
- *Generator action* (The generator adds the standard key struct as a parameter.)

Non-standard remove (generated SQL)

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

Non-standard remove description

Non-standard remove operations delete records from the database table of the parent entity that matches the information in a key struct that you provide.

Non-standard remove use

Based on a key that you specify, a *non-standard remove* operation to delete a database record or records.

If the key that you specify is not unique, multiple database records are deleted.

As with any operation where you specify the key, there is no guarantee that the database can access the target records efficiently. Instead, the designer must define appropriate indices to ensure that the database can access the target records efficiently.

Non-standard remove 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 you use segmented tablespaces with DB2 for z/OS (the default for version 9), IBM changed the behavior of the JDBC driver. For more information, see the *TechNote on JDBC executeUpdate method can return a negative row count when the database server is DB2 for z/OS* related link. Therefore, a `RecordNotFoundException` error is thrown when a negative row count is returned (that is, a `DELETE FROM` with no predicate).

- *Parameters* (A non-standard key struct to specify the record to modify. This non-standard key can specify multiple records. In this case, all records that match the non-standard key are deleted.)
- *Return Value* (None.)
- *Generator action* (None.)

Related information

JDBC executeUpdate can return a negative row count when the database server is DB2 for z/OS

Non-key remove

A non-key remove operation has a stereotype `nkremove`.

Non-key remove 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 execute SQL statements that do not have a where clause, that is, they operate on all rows on a table.

For a non-key read operation, there must be a single row on the table. Typically, use this type of operation to read a value from a control table that contains a single record.

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

Non-key remove use

Use a *non-key remove* operation to delete all records from a database table.

Non-key remove parameter and generator notes

Non-key remove operations do not take parameters.

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

1.19 Entity batch operations

Batch operations insert or remove many records from the database.

The following are the two batch operation types:

- `batchinsert`
- `batchmodify`

Related concepts

batchinsert

A batch insert operation has a stereotype `batchinsert`.

batchinsert description

Use batch insert operations to insert many records into the database. When you batch operations together, the database is required to perform fewer round trips and it improves performance.

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. Instead, the insert statement is added to a batch of statements that are stored locally by the Cúram infrastructure by calling the `java.sql.PreparedStatement.addBatch` method. Once the batch reaches the desired size, it must be executed by calling the `$execute` method of the operation.

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

As a result, you cannot spread batched inserts or modifies across multiple client invocations in an online environment as 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 that 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.

For more information about this array and how queued statements are executed, see the JDK documentation for `java.sql.PreparedStatement`.

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. The optimal size can be different for each individual batch operation. You or the DBA determine the optimal size.

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, simply call the `$execute` method of the operation, and then continue as before.

batchinsert use

Use a `batchinsert` operation to insert many records to the same entity in a single transaction.

batchinsert parameter and generator notes

A `batchinsert` operation takes a single input parameter, that is, a structure that defines the attributes to insert. Each attribute of this structure must match some entity attribute by name and type.

The system displays a warning if a `batchinsert` operation contains a non-standard details parameter that does not include fields that cannot be null. For more information, see the *Null considerations* related link.

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

Generator Action. The generator adds the following methods to a class that contains a batch insert operation:

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

Related reference

“Null” considerations on page 81

When you write a handcrafted SQL statement, some Cúram datatypes are stored as null on the database if they are empty (that is, in their initial state).

batchmodify

A batch modify operation has a stereotype `batchmodify`.

batchmodify description

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

batchmodify use

Use a batch modify operation to modify many records on the same entity in a single transaction.

batchmodify parameter and generator notes

A batch modify operation uses non-standard key and details structures as input parameters. You must create the input parameters 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. You can also use standard (generated) key or details structures.

A warning is displayed if a `batchmodify` operation has non-standard details parameter, which does not include fields that cannot be null. For more information, see the *Null considerations* related link.

- *Parameters* (A non-standard key struct to specify the record to modify. This non-standard key can specify multiple records. In this case, all records that match the non-standard key are updated.)

  A non-standard details struct that contains the updated version of the data.
- *Return Value* (None.)

Generator action (Use the generator to add the following methods to a class that contains a batch modify operation:

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

Note: You cannot spread `batchmodify` operations across multiple client-server invocations (transaction). You can only use a `batchmodify` operation in an online transaction if the batch is executed before the end of the transaction.

Related reference

"Null" considerations on page 81

When you write a handcrafted SQL statement, some Cúram datatypes are stored as null on the database if they are empty (that is, in their initial state).

1.20 Entity handcrafted SQL operations

Use non-standard (ns) operations with handcrafted SQL against the database.

The following are the two types of ns operation:

- `ns`
- `nsmulti`

Non-standard

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

Non-standard description

Parameters for a non-standard (ns) operation must be structs. Ns operations must be structs because parameters are replicated in the Data Access Layer (DAL) and the DAL only allows parameters to be structs.

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

You *must* provide SQL with all the ns operations. No SQL is automatically generated.

Ns operations must belong to an entity class. However, the SQL query can operate on any database table. The SQL query does not have to operate only on the database table that belongs to the entity class, that is, you can use it to perform SQL joins across tables.

For information on how to specify SQL in an operation, see the *Using handcrafted SQL in non-standard entity operations overview* related link.

Related reference

Using handcrafted SQL in non-standard entity operations overview on page 81

Non-standard use

Use a non-standard operation for a database operation that is too complex for any of the preceding operations and that does not retrieve multiple records.

The following are examples of where you use the non-standard operation:

- Queries whose where clause contains comparisons other than equals, such as less-than, greater-than, and so on.

- Queries or commands that operate on more than one database table.
- Queries that return something other than an attribute of a table, such as the results of `max` and `count` functions

You must specify the SQL to execute and you can specify zero or many parameters for the operation. All parameters must be structs and must be flat, that is, they cannot aggregate other structs.

The handcrafted SQL can perform any database operation so long as a cursor is not required. This includes single-record-reads, single or multiple record updates and deletes, and joins across multiple database tables as the parameter structs cannot aggregate other structs. If your handcrafted SQL requires a cursor, then use a `nsmulti`operation.

Non-standard parameter and generator notes

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

Non-standard multi

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

Non-standard multi description

Typically, non-standard multi operations are similar to non-standard operations.

The following restrictions account for the differences between non-standard multi operations and non-standard operations:

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

Typically, you use non-standard multi operations for performing readmulti operations that join two or more database tables.

A non-standard multi operation is the only entity operation that cannot use additional parameters. Usually, this is 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.

Non-standard multi use

Use non-standard multiple operations in two specific circumstances.

Use *non-standard multiple* operations in either of the following circumstances:

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

A non-standard multiple operation is 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 execute. This means you can reference multiple database tables and/or to specify complex where clauses.

Non-standard parameter and generator notes

You cannot specify any extra parameters for the non-standard parameter operation.

- *Parameters* (Key parameter [optional].)
- *Return value* (List-details parameter.)
- *Generator action* (The generator creates a list wrapper for the return-value struct you specify and uses this as the return value for the operation.)

Example 1: nsmulti with a single (list) parameter

The proceeding is an example of an operation in the input meta-model to list every transaction in the system where the amount is for less than one dollar.

The following struct is defined in the model and contains the information about each transaction. The type of each attribute of the struct is not relevant here and is omitted for clarity.

- struct class: `MinorTxDetails`

Attribute	Domain
txDate	DATE
txAccountNumber	ACCOUNT_NUMBER
txAmount	AMOUNT

The proceeding table shows an entity that is defined in the model with some of the attributes that the `nsmulti` operation `getMinorTransactions()`, returning an instance of `MinorTxDetails`, would use.

- entity class: `BankAccount`

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

The proceeding is the SQL for the operation (which you supply in the model):

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

Figure 8: SQL for nsmulti with a single (list) parameter

You provide all of the SQL. The generator produces the remainder and the proceeding shows this for illustrative purposes.

The following pseudo code describes the structs that this operation uses. The actual Java structs corresponding to the structs that are 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;
}
```

Figure 9: Pseudocode for generated structs for use by nsmulti operation

The code generator produces the Java interface for this entity class, complete with the `nsmulti` operation. The proceeding shows how it would look:

```
          public interface BankAccount {

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

Figure 10: Generated Java interface for nsmulti operation

The preceding demonstrates how you would write handcrafted Java code to call this method and to iterate through each element that is 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;
}
```

Figure 11: Calling a nsmulti operation from handcrafted Java code (one parameter)

Example 2: nsmulti with two parameters (key + list)

In the proceeding example, instead of returning all transactions for less than one dollar in the whole system, it returns only the transactions *for one account* that were less than one dollar.

Another parameter is required to specify the required account number. Since nsmulti is a database operation and database operations require all parameters to be structs, use a struct for the account number parameter even though the struct has only one field.

Note: The account number field appears in various guises - txAccountNumber, txAccountNum, and theAccountID. Unlike the other database operations, the names of attributes are not required to correspond when you use ns or nsmulti operations as the handcrafted SQL can reference the different field names as appropriate

- struct class: `AccountNoWrapper`

Attribute	Domain
txAccountNumber	ACCOUNT_NUMBER

You can now use this struct as an input argument to the nsmulti operation `getMinorTransactions(theAccountID : AccountNoWrapper)`, returning an instance of `MinorTxDetails`, for the proceeding entity:

- entity class: `BankAccount`

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

The proceeding is the SQL for the operation:

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

Figure 12: SQL for nsmulti with a key and list parameters

This is all you must provide. The generator produces the remainder and, for illustrative purposes, the proceeding shows this.

The proceeding pseudo-code describes the structs that the operation uses. (The code generator produces the actual Java structs corresponding to the structs defined in the model.)

```
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;
}
```

Figure 13: Pseudocode for generated structs for use by nsmulti with key and list parameters

The code generator produces the Java interface for this entity class, complete with the nsmulti operation. It would look like the proceeding:

```
          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;
};
```

Figure 14: Generated Java interface for nsmulti operation with key and list parameters

The proceeding demonstrates how you write handcrafted Java code to call this method and to iterate through each element that the method returns:

```
<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;
}
```

Figure 15: Calling a nsmulti operation from handcrafted Java code (two parameters)

Using handcrafted SQL in non-standard entity operations overview

For entity operations of stereotype `ns` and `nsmulti`, you must specify the SQL to use 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.

Using host variables

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

The following are the rules for using host variables:

- Prefix host variables with a colon (:).
- Ensure that 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 use the role name of aggregation for the host variable.

For example:

:dtls

For more information about aggregation, see the *One-to-one aggregation* related link.

Related reference

One-to-one aggregation on page 85

One-to-one aggregation embeds a single instance of one class within another.

“Null” considerations

When you write a handcrafted SQL statement, some Cúram datatypes are stored as null on the database if they are empty (that is, in their initial state).

For this reason, when you search for these records your query must search for “null” rather than an empty string. The following examples explain using a query to search for “null” rather than an empty string.

Incorrect "null" search

The following is an example of an incorrect "null" search.

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

Correct "null" search

The following is an example of a correct "null" search.

```
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. In this case, the Allow NULLs on this database field option defaults to `no`. If necessary, you can override this default.

Note: The Allow NULLs on this database field option controls the NOT NULL qualifier in generated DDL in an inverted way. If you set this option to `no`, it adds the NOT NULL qualifier; if you set it to `yes`, it omits the qualifier.

Table 1 shows the Cúram data types that can be represented as a null on the database.

Table 8: Data types and nulls

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

Update considerations with DB2 for z/OS

If you are running against a DB2 for z/OS database, you might need to modify any handcrafted SQL that explicitly uses a `FOR UPDATE` clause to prevent the system from throwing `RecordLockedException` errors.

If the particular SQL statement is invoked simultaneously by multiple users, 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.

SQL example 1

In the following example, entity `Employer` has a method `CountEmployers` (`stereotype ns`) that returns the number of records in the `Employer` table.

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

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

Figure 16: Struct for return result

The following extract is the Java interface for this operation:

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

Figure 17: Java Interface

Finally, the SQL to implement this query is:

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

Figure 18: SQL Implementation

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

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

SQL example 2

The following example shows how to use parameter host variables and expands the previous example by adding another method that 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; }
```

Figure 19: 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 = "";
}
```

Figure 20: Struct for employer key

The SQL statement for this method is:

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

Figure 21: SQL Implementation

Note: As the second parameter contains *longValue*, it is necessary to qualify it with *2.*. Unqualified parameter references are assumed to reference the first parameter.

The SQL statement in Figure 4 qualifies both parameters and is equivalent to the SQL statement in Figure 3:

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

Figure 22: SQL Implementation with qualified parameters

1.21 Aggregation

Use aggregation to embed or nest instances 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.

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

The generator permits the following aggregation configurations:

- Structs can aggregate structs.
- Structs can aggregate entities.

A special case

The generator supports the aggregating of standard details structs, even though standard details structs do not appear in the input model.

Standard details structs are aggregated by aggregating the entity class that “owns” the standard details struct.

One-to-one aggregation

One-to-one aggregation embeds a single instance of one class within another.

The following example describes how to aggregate a struct class, `PersonDetails`, into to another struct class, `PersonDetailsWrapper`, by using one-to-one aggregation.

To create a one-to-one aggregation, create a Rational® Software Architect Designer diagram and then complete the following steps:

- Add classes `PersonDetails` and `PersonDetailsWrapper` to the diagram.
- In the diagram, drag the appropriate arrowhead (it 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 pop-up menu.
- Select the aggregation relationship in the diagram.
- Open the **General Properties** tab.

The preceding steps create the aggregation relationship whereby one role corresponds to class `PersonDetailsWrapper` and the other role 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, has its aggregate option set. The assignment of Role A and Role B is arbitrary. The key thing is that the role with the checked aggregate box denotes the outermost class of the pair.

When you select the relationship line in the diagram, the **General Properties** tab displays `PersonDetailsWrapper` in the graphic at the top of the properties sheet with the diamond that is associated with it. Set the following properties of the aggregation:

- The label is optional.
- For `PersonDetailsWrapper`:
  - Indicate **Composite** in the aggregation radio button.
  - Set **Multiplicity** to 1.
- For `PersonDetails`:
  - Indicate **None** in the aggregation radio button.
  - By default, the role is set to **dtls**.
  - Set **Multiplicity** to 1 (to signify a one-to-one aggregation).

The class diagram appears in Rational® Software Architect Designer. The diagram shows the two classes that are joined by the UML aggregation relationship line (with the diamond end touching `PersonDetailsWrapper`). Each side of the relationship shows multiplicity of one and the `PersonDetails` shows 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 that is produced from this construct takes 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();
};
```

One-to-many aggregation

One-to-many aggregation embeds a sequence of one class within many others.

This example models a one-to-many aggregation, which means that a list of one class type is embedded into the other class. Start by creating `PersonDetailsList`, which aggregates a list of `PersonDetails`. To create a one-to-many aggregation, open a Rational® Software Architect Designer diagram and complete the following steps:

- 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 pop-up menu.
- Select the aggregation relationship in the diagram.
- Open the **General Properties** tab.

The preceding steps create the aggregation relationship where 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.

When you select the relationship line in the diagram, the **General Properties** tab displays `PersonDetailsList` in the graphic in the properties sheet with the diamond that is associated with it.

Set the following properties of the aggregation:

- The **Label** is optional.

- For `PersonDetailsList`:
  - Indicate **Composite** in the aggregation radio button.
  - Set **Multiplicity** to *.
- For `PersonDetails`:
  - Indicate **None** in the aggregation radio button.
  - By default, the role is set to **dtls**.
  - Set **Multiplicity** to 1..* (to signify a one-to-many aggregation).

The class diagram appears in Rational® Software Architect Designer. The diagram displays the two classes that are joined by the UML aggregation relationship line (diamond-end touching `PersonDetailsList`. The diagram displays the aggregates side of the relationship that shows a multiplicity of `*` and `PersonDetails` showing a multiplicity of `1..*` and a role name of `-dtls`.

The pseudo-code that is produced from this construct takes 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(final 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` includes a field that is named *dtls*. This field provides functionality that is needed for lists such as adding items, getting an item by index, and getting the list contents as an array.

1.22 Assignable

Use generated struct classes to automatically assign values between matching fields in another struct as provided by the generated struct'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 that are shared with the `BankBranchStruct` class:

- *bankBranchID*
- *bankSortCode*
- *name*

The following Java code illustrates how to create these objects:

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

Typically, the following illustrates the assignment from one struct to the other:

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

To simplify the preceding code, use the `assign` function, which becomes more significant as the size of the structs increase:

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

Use an assignable relationship to allow further control of the specifics of the automatic assignment with the `assign` function. It is required when you want to do explicit field assignment between fields with differing names or to suppress the default assignment between fields of the same name.

Explicit field assignment

An explicit field assignment is where fields with different names are matched.

An explicit field assignment is represented in the model by adding an assignable relationship between the two classes, and then adding attributes to be matched to both sides of the assignment. Any fields that are not explicitly linked are treated as default assignment fields.

This is shown in the following classes:

- entity class: `Address`

Attribute
addressID
addressLine1
addressLine2
addressLine3
addressLine4
cityCode
countryCode
postalCode
regionCode
comments

- struct class: `BankBranchStruct`

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

In an assignable relationship between the two classes `Address` and `BankBranchStruct`, you can explicitly map fields. For example, match `BankBranchStruct.cityID` with `Address.cityCode`. In Rational® Software Architect Designer, mapped fields are 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. The generator automatically handles all the other common fields (for example, *AddressLine1*, and so on).

For instance, the following illustrates the generated code without the explicit field assignment:

```
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 is 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, you can reduce these lines of code to just one line:

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

Suppressing default assignment fields

In some situations, you might not want to match a pair of similarly-named fields. You can omit a pair of fields from an assignment by listing one of the fields at one end of the relationship.

For the proceeding two classes, `PersonInfo` and `AccountInfo`, with a 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, first create the objects for the `PersonInfo` and `AccountInfo` classes as (see the preceding steps):

```
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. Now this assignment:

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

is equivalent to the following *two* statements (that is, the *Id* assignment is no longer made):

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

Combining structs

In some instances, you might need to populate one struct with the contents of two or more other structs.

Figure 1 illustrates a typical piece of Java code.

```
        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;
```

Figure 23: Example Java code for combining structs

By explicitly mapping the *BankBranchStruct.addressVersionNo* attribute to the *Address.versionNo* in the assignable relationship, you can now write the Java as:

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

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

Figure 24: Equivalent Java code for combining structs

Note: In this case, the second `assign` does not overwrite the first as it references a different subset of fields. As a result, the effect is to merge the two struct contents.

1.23 Foreign keys

Use the Cúram generator to create foreign keys 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 name the relationship. The relationship name is then applied to the foreign key constraint added to the database. Otherwise, the database chooses its own name for the constraint.

The following rules apply to using foreign keys:

- Foreign key relationships are allowed on entity classes only.
- Fields that are referenced by a foreign key are set to unique, as the unique field is required by some databases.
- If the foreign key references the primary key of another entity, the generator does not produce a redundant unique clause as the primary key is already unique.
- Foreign keys cannot be specified on subclass entities. You must specify the relationship by using the actual base entity classes themselves.

Adding 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 by adding key/qualifiers to the role that is touching the *referenced* class.

Specifying a foreign key in this way is represented on a class diagram by a line between two classes, with a box that contains 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 shows two classes that are 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.

Primary and foreign key naming constraints

You can include a constraint name for foreign key constraints in Cúram models.

The name that you give in the model to the `foreignkey` relationship is applied by the system to the foreign key constraint itself. If necessary, you can suppress this feature 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 its corresponding entity. This also results in an accompanying index of the same name. You can suppress this feature by specifying `-nonamedprimarykeyconstraint` on the generator command line.

Foreign key example

This foreign key example uses two entity classes.

Consider two entity classes, `BankAccount` and `BankTransaction`, where `BankAccount`. *accountNo* is a `foreignkey` on `BankTransaction`. This means that the BankTransaction table (txAccountNo) must have a record on the BankAccount table with a matching accountNo value.

The following table illustrates two classes where the `foreignkey` is 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
details txID	TX_ID
details transactionDate	TX_DATE
details transactionType	TX_TYPE
details transactionAmount	TX_AMOUNT

This foreign key generates the following DDL (Oracle SQL shown):

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

1.24 Indexes

Use the Cúram generator to create indexes other than the primary index on database tables.

You can create any number of indexes on each table, with the usual speed versus database size trade-offs that are associated with indexes.

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.

Using a struct to represent an index does not have any side-effects on the struct (apart from the preceding caveats). You can still use the struct as an argument to an operation. Typically, you use the struct as both a key parameter and as an index to support database accesses via this key.

The following rules apply to using indexes:

- You must name the relationship. The database uses the name to index. The name of the struct in the relationship does not affect 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.

Adding an index to a database table overview

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 unimportant.

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

Index naming overview

You never explicitly reference an index, but referencing an index is a requirement for the Database Administrator.

As you never explicitly reference an index, use index names that are as meaningful and descriptive as possible.

Index example

This example uses two classes with an index relationship.

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

- entity class: `BankClient`

Attribute	Domain
key clientID	CLIENT_ID

Attribute	Domain
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 preceding index produces the following DDL by the generator:

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

1.25 Unique indexes

To model a unique index, add a unique index stereotype relationship between an entity class and a struct class. The rules for modeling a unique index are the same as the rules for modeling a non-unique index.

When you specify a unique index for an entity, the necessary information is included in the generated file *<Application-name>_unique_constraints.xml*. You must then reference this file from the data manager configuration file (*datamanager_config.xml*).

Note:

The file *<Application-name>_unique_constraints.xml* contains two sets of information:

1. Unique indexes 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 are implicit unique constraints that are automatically produced by the generator and that are applied to all fields that 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 that are 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 you run the data manager, it creates the explicit unique indexes before the implicit unique constraints. By doing this, it means that the database can use the developer-specified unique indexes to enforce uniqueness rather than having to create and use its own system-named indexes.

For example, you might want to model a specifically-named unique index to correspond to a particular foreign key in the model. The generator automatically gives the unique constraint the same name as the corresponding unique index.

1.26 Generated class hierarchy

A hierarchy of classes that are generated by the server-code generator correspond to the classes designed in the application model.

All classes are defined in the Cúram model by using UML notation. A single process, facade, or entity class can contain a mixture of automatically generated and handcrafted methods. Do not store handcrafted code and generated code in the same file due to the risk that the generator overwrites handcrafted code or vice versa.

Code that you provide is stored in a single class. The code is generated into a number of other classes and the set is linked together into a hierarchy by inheritance and implementation.

Note: As struct classes do not contain operations, you do not need to separate handcrafted and generated code. Each struct class in the model corresponds to one generated Java struct class.

Basic hierarchy example

This example shows the elements of the generated and required handcrafted class hierarchy for a basic entity class named `MyClass`. This class does not use inheritance or code packages.

The UML representation of the generated Java classes of the entity class `MyClass` shows 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.

So, four Java classes correspond to the entity class in the UML model. Of the four classes, three share the name as the class in the model. The fourth class shares the name with the word `Factory` appended.

The following provides a further description of the classes:

1. `<ProjectName>.intf.MyClass`

   This is a generated Java interface class that contains all the public methods for the class.

   The other classes in the hierarchy, either generated or handcrafted, are required to provide implementations for these methods.
2. `<ProjectName>.base.MyClass`

   This is a generated abstract Java class that implements the interface in the `intf` version of the file. The following is contained in the file:

   - The implementations of data access methods (that is, stereotyped methods of entity classes) and connector methods.
   - Abstract method declarations for exit point methods.

     This is to ensure that you are forced to provide implementations for the exit points.
   - Abstract method declarations for methods declared protected in the model.

     This is to ensure that you are forced to provide implementations for these methods without exposing them in the interface (`intf` layer) for the class.
3. `<ProjectName>.impl.MyClass`

   You supply this class and the class always inherits from the corresponding `base` version.

   You must declare it abstract to ensure that the class cannot be instantiated directly. The class must only be instantiated by using the factory mechanism - see the proceeding step.

   In this class, you 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 to avoid the risk of either your code overwriting generated code or vice verse.
4. `<ProjectName>.fact.MyClassFactory`

   This is a generated Java class that contains one static method: `newInstance()`. This method creates instances of the class and is the only way to instantiate entity, facade, and process classes.

   As a factory creates all instances of objects, you can also use it for the following:

   - Transparently create and return a customized version of the class requested. For more information, see *Replacing the superclass* related link. You do not need to change pre-existing code that you used in the original version of the class.
   - Transparently create and return a proxy class of the requested class. The proxy class wraps the requested class (by 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 to create an instance of `MyClass`.

Note: 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();
```

Figure 25: Using a factory to create an instance of MyClass

Related reference

Replacing the superclass on page 113
When you define a subclass, you can specify that the subclass replaces its superclass entirely.

Subclasses hierarchy example

This example shows the elements of the generated or 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.
  - It contains any required (non-generated) handcrafted methods.
- `<PackageName>.fact.SubClassFactory`
  - A subclass of the `impl` class.
  - It returns an instance of the `intf` class.

As with the basic hierarchy example, there are four Java classes corresponding to class `SubClass`. However, as `SubClass` inherits from `MyClass` it creates two extra relationships. The extra relationships are as follows:

1. Interface `SubClass` inherits from interface `MyClass`. This ensures that `SubClass` must implement all of its own declared methods as well as the methods 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. Therefore, these methods are available to `SubClass` and you are not required to re-implement them.

Abstract classes hierarchy example

In a Cúram model, you can mark classes abstract. This means that the classes cannot be instantiated.

For more information on abstract classes, see the *Entity class options* related link.

From the subclasses 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.
  - It contains any required (non-generated) handcrafted methods.

The hierarchy is the same as for non-abstract classes except that it generates no factory.

Related reference

Entity class options on page 30
The options available for entity classes are entity class abstracts, allow optimistic locking, audit fields, enable validation, last updated field, No Generated SQL, and replace superclass.

Class hierarchy considerations

The following considerations apply to class hierarchies: access control, the meaning of super, and enforcing the factory mechanism.

Public and protected access control

In Cúram models, the Java language supports two levels of access control: public and protected.

Typically, the Java language supports four levels of access control for methods and member variables: private, protected, public, and package. As the generated class hierarchy in Cúram includes different classes in different packages, there is no need to use the private and package access levels.

Note: Not using private and package access levels only applies to operations in Cúram models.

You can still use private and public access in handcrafted Java code as required.

The meaning of super

In Java, the super keyword is a reference to the superclass, that is, the class from which the current class (this) inherits.

In the subclasses hierarchy example, the superclass of `SubClass` is `MyClass`. However, when you write handcrafted Java code for `<ProjectName>.impl.SubClass` note that the superclass of this class is `<ProjectName>.base.SubClass` rather than any version of `MyClass`.

Related reference

Subclasses hierarchy example on page 98
This example shows the elements of the generated or handcrafted class hierarchy for a basic entity class named `SubClass` that inherits from `MyClass`.

Enforcing the factory mechanism

To create entity, facade, and process objects, only use their associated factory classes.

Do not bypass the associated factory mechanism by using the new keyword to instantiate these classes. Enforce this by making all implementation classes (that is, all classes in the `impl` packages) abstract. If you do not make these classes abstract, there is a risk of instantiating these classes directly and, consequently, class replacement not working as expected.

Class hierarchy summary

Specific individual objects in a Cúram application model appear as multiple classes in the output code.

Ensuring the following is the objective of the generated class hierarchy:

- You provide 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.
- You are 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.
- To support replacement and tracing the runtime type of an object is determined by a factory.

1.27 Cúram JMS queue connectors

Use Cúram connectors in a Cúram application to connect to other systems through JMS queues.

For facade and process class operations with a stereotype of qconnector, the generator produces code that:

- Converts the operation parameter into a JMS message.
- Places the message on a queue and optionally waits for another message.
- Converts the message back to a Cúram struct.
- Returns the struct to the caller.

For many operations, you can implement queue connectors without writing any handcrafted code. You can also customize connectors with the use of handcrafted code. In the following instances, customize connectors with the use of handcrafted code:

- The default encoding of a datatype is not suitable for your purpose. For example, you want to encode dates in the form `DD-MMM-YYYY` instead of the default format of `YYYYMMDD`.
- Your parameter struct is “complex”. For example, it contains a variable length field or aggregates another struct.

JMS queue connectors overview

Use a connection factory to build connections.

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. Each field 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 you specify a return type for the operation, the Cúram application waits for a response message. Typically, it waits 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.

qconnector operation options

Use the appropriate qconnector operation.

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. Instead, connection factories create queue connections. 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. It specifies the JNDI name of the queue from which the response message is taken.

- **Message Type**

  `BytesMessage` or `TextMessage`. Use this message type to specify whether a JMS `BytesMessage` or `TextMessage` is sent or 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 or systems you are communicating with use a different character encoding, then these bytes might be incorrectly translated by the other systems. In this case (provided the message doesn't contain any binary data), use a JMS `TextMessage` to ensure that the message is correctly translated by the other systems.
- **BytesMessage encoding character set**

  This 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 you do not specify this option, then the default local system character encoding is used. (Usually, this is 'Cp1252' for Microsoft® Windows, 'Cp1046' for EBCDIC on IBM® z/OS® , and so on.) Use this to ensure that the character encoding for the message matches the character encoding of the other system with which you are communicating.

  This option is not relevant if you use `TextMessage`.

qconnector operation considerations

qconnector operations are represented in the meta-model and implemented on the remote system.

Determine the message format or formats and create corresponding struct or structs

With the application developer, agree on the format or formats of the messages that pass between the two systems.

Agree the following with the application developer:

- **The format of each field in the message**

  You can use the default encoding method for each field. For information on how to implement a custom encoding methodology, see the *Encoding methods for fundamental types* related link and the *Using customized encoding and decoding classes* related links.
- **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. For information on lengths of datatypes, see the *Encoding methods for fundamental types* related link. You can change the length of the field 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. Use the toolbar to change the order of struct attributes, if required.

Related reference

Encoding methods for fundamental types on page 104

Use the table to view the datatype and encoded width for each encoding method.

Customized encoding and decoding class usage on page 105
By default, the encoding method for each field in a struct that is used or returned by connector operations is based on the field type.

Add the operation to the application meta-model

A qconnector operation is modeled like any other process or facade class operation and is subject to restrictions.

For more information on the restrictions, see *qconnector rules and restrictions* related link. To specify the queue or queues and some queuing parameters, you must use some operations. For more information on the operations, see the *qconnector operation options* related link.

In summary, the method must contain one struct parameter, it might return void or a struct, and must contain options to identify the MQSeries queues to use.

Related reference

qconnector rules and restrictions on page 103
The qconnector is subject to specific rules and restrictions.

qconnector operation options on page 101
Use the appropriate qconnector operation.

Configure the queues in the application server

The queue connection factory and references to the queues themselves are stored in the JNDI namespace.

Map these JNDI names to actual connection factories and queues in the application server configuration.

Implement the message recipient in the remote system

The message recipient can be any system that has access to the MQSeries queues.

Typically, the message recipient is a legacy system that the Cúram application requires access. 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, that is, if you specified a return type for the operation, then the remote system must:

- Create a response message AND
- Send the response message back to the waiting Cúram application.

The response message is associated with the original message by using its *CorrelationID*, that is, *the message recipient must set the CorrelationID of the response message equal to the MessageID of the original message.*

qconnector rules and restrictions

The qconnector is subject to specific rules and restrictions.

The following rules and restrictions apply to qconnectors:

- The qconnector operation stereotype is valid in process or facade classes only.
- Connector operations must have exactly one struct parameter.
- Connector operations can have a return type of void or a struct.
- The parameter and return structs can take any form. However, the code that you generate can only map structs that are “flat”, that is, structs that do not aggregate other structs and that contain only fixed-length fields.

  For complex structs, you must implement a mapper class to map the struct to and from messages. For examples of coding and decoding complex structs, see the *Encoding methods for fundamental types* related link.

  .

Related reference

Encoding methods for fundamental types on page 104

Use the table to view the datatype and encoded width for each encoding method.

Encoding methods for fundamental types

Use the table to view the datatype and encoded width for each encoding method.

Table 9: Encoding methods

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)
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

Datatype	Encoded Width	Encoding method
SVR_UNBOUNDED_STRING	N/A	Not natively supported

- 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 permitted amount, 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 as the string length is not known when the string is generated and is required for creating fixed-length messages. However, it is possible 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`.

Customized encoding and decoding class usage

By default, the encoding method for each field in a struct that is used or returned by connector operations is based on the field type.

For example, the mapper class for `curam.util.type.DateTime` is `curam.util.connectors.mqseries.MQFieldMapper.DateTimeMapper`. Similarly, for boolean fields it is `curam.util.connectors.mqseries.MQFieldMapper.BooleanMapper`.

For any individual field in any operation, you can override this default and specify the name of the class to map the data. You specify the names of the custom mapper classes in the properties file *`QueueConnectorFieldMappers.properties`*, which you must include in the application classpath.

Use the following format for entries in the properties file:

[class].[operation].[param].[field]=[mapper]

where

- [class] is the name of the process or facade class that contains 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. [mapper] 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
```

Figure 26: Sample QueueConnectorFieldMappers.properties

Working with variable length fields example

The following example uses a custom field mapper class to implement a primitive variable length field message.

Encode the variable length field by prefixing the data with a six character string that contains a number that specifies the length of the data in the remainder of the string.

Note: The following example only shows the implementation at the Cúram end of the queue. The remote system also needs to recognize the encoding method and implement the necessary translations by using the language of choice on the remote system.

The following pseudo code describes the struct that is used in the operation. Fields *idNumber* and *dateOfBirth* use the default conversion methods for their type and are converted into 10 and eight character strings, respectively. The *historyText* field is a variable length field and is encoded and decoded by using a custom mapper class.

```
struct PersonHistory {
  String<10> idNumber;
  String historyText;
  Date dateOfBirth;
}
```

Figure 27: Pseudo code for the struct to be mapped:

Use method `addToHistory` of class `LegacyBPO` to send a `PersonHistory` struct to a legacy system. The legacy system appends text to the variable length field *historyText*, and returns an updated copy of `PersonHistory`.

```
interface LegacyBPO {
  PersonHistory addToHistory(dtls PersonHistory);
}
```

Figure 28: Pseudo code for the BPO interface

Note: Field *historyText* is 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
```

Figure 29: The property file entries that link the fields to the mapper

Figure 4 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;
  }
}
```

Figure 30: Mapper class implementation for variable string

Examples of MQSeries messages that are 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 eight characters are the *dateOfBirth* field.
- The middle section is the variable length *historyText* field, of which the first six characters specify the length of the data.

Working with lists example

The following example uses a custom field mapper class to implement encoding and decoding of a struct that aggregates a list of another struct.

The list is encoded into a single string. The first four characters contain a number that specifies the number of entries in the list. The remainder of the string consists of the encoded form of each struct as a fixed-length string. This example illustrates how list aggregations are handled when aggregations implement a custom mapper class.

Like the variable length fields example, this example only shows the implementation at the Cúram end of the queue. The remote system also needs to recognize the encoding method and implement the necessary translations by using the language of choice on the remote system.

The following pseudo code describes the struct that is used in the operation. Struct `PersonDtls` is encoded as a fixed length 18 character string. Struct `PersonDtlsList` are encoded by:

- Encoding each struct in its list AND
- Concatenating the results into a string AND
- Prefixing the string with a six character string that specifies the number of entries in the list.

```
struct PersonDtls {
  String<8> idNumber;
  String<10> surname;
}

struct PersonDtlsList {
  sequence <PersonDtls> dtls;
}
```

Figure 31: Pseudo code for the structs to be mapped:

Method `processNames` of class `LegacyBPO` sends a `PersonDtlsList` struct to a legacy system, the legacy system performs some processing on this data, and returns an updated copy of `PersonDtlsList`.

```
interface LegacyBPO {
  PersonDtlsList processNames(p1 PersonDtlsList);
}
```

Figure 32: Pseudo code for the BPO interface

Again, as in the previous example, field *dtls* of struct `PersonDtlsList` is 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 are split for clarity).

```
LegacyBPO.processNames.p1.dtls=
  com.acme.mqutils.PersonDtlsListMapper
LegacyBPO.processNames.return.dtls=
  com.acme.mqutils.PersonDtlsListMapper
```

Figure 33: The property file entry that links the fields to the mapper

The following 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;
  }
}
```

Figure 34: Mapper class implementation for list of structs

For example, the following list of `Ent18131` structs:

- `("0000361i", "James")`
- `("0024684x", "John")`
- `("8211519f", "Sharon")`

are encoded as follows:

```
       "3   0000361iJames
 0024684xJohn      8211519fSharon    "
```

Where:

- The first four characters contain a number that specifies the number of encoded structs to follow AND
- The remaining string consists of three 18 character blocks corresponding to the three encoded structs.

Related reference

Working with variable length fields example on page 106
The following example uses a custom field mapper class to implement a primitive variable length field message.

1.28 Subclass modeling

Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

You cannot use subclassing to add extra attributes to entities or structs.

The following are examples of when you would subclass a class:

- Adding new stereotyped methods to an existing entity class.
- Adding or contributing to an existing entity or operation exit points.
- Modifying existing entity operation Readmulti Max options.

Basic subclassing

To transform a class into a subclass, add a generalization relationship from the subclass to the superclass (base class).

On a class diagram, the generalized relationship displays as a line between the two classes with an arrow that points toward the superclass. Therefore, the subclass inherits all the operations of the superclass. In addition, it might:

- Add extra functions.
- Modify the applicable options of the function in the superclass.

In the following 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.

Replacing the superclass

When you define a subclass, you can specify that the subclass replaces its superclass entirely.

To turn on the feature for an individual entity class, set the Replace_Superclass property in the Rational® Software Architect Designer Curam Properties tab to `1 - yes` by using the supplied drop-down.

For example, setting Replace_Superclass to `yes` for a class, `MySubclass`, means that instances of the base class, `MyBaseClass`, are no longer be created. All requests for the base class (`MyBaseClass`) now receive an instance of the subclass (`MySubclass`). The factory mechanism handles the request and the request is transparent.

Abstract classes

To make a class abstract, set the class's Abstract option to `yes` in the meta-model.

When you set the class's Abstract option to `yes` in the meta-model, a factory class is not included in the generated Java class hierarchy for this abstract class. As a result, the class cannot be instantiated. The purpose of the class is to enable it to be subclassed.

All non-abstract subclasses of the abstract classes contain the factory component and are instantiated in the normal way.

You must provide the `impl` Java code for abstract classes (unless the abstract class has no subclasses). From this point, 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 no implementation is provided must be implemented by the subclass or subclasses.

Restrictions

Cúram generators do not support multiple inheritance. You can only use subclassing to add or override operations - you cannot use subclassing to add or override attributes.

Writing code for subclassing

No specific restrictions apply to writing code for subclassing. You can subclass any entity, facade, or process class without changing how you declare or use the class.

Recommendations for writing code for subclassing

- Write new subclasses of existing classes in new source files.
- The generated class hierarchy dictates the packaging of the new source files.
- Place all new source files within the source subdirectory of the *`EJBServer\components\<custom>`* directory, where:
  - <custom> is any new directory that is created under the components directory that conforms to the same directory structure as *`components\core`*.

Using subclassing to override validation exit points example

To override the validation exit point of an entity in a subclass, perform two steps.

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.

In the example of two classes, `MyEntityClass` and `MyEntitySubClass`, the subclass `MyEntitySubClass` would inherit the key and the details of the superclass. For the subclass `MyEntitySubClass`, the Automatic validation operation option is enabled and would add the `insert` or `modify` operations.

For more information about validation exit points, see the *Validation* related link.

Related reference

Validation on page 35

The validation function is called before standard insert and standard update operations, and also before the pre-data access functions. It provides a common place to put validation code.

Overriding Pre Data Access, Post Data Access, and On-Fail exit points example

To override the Pre Data Access, the Post Data Access, or the On-fail exit points of an entity in a subclass, perform two steps.

To override the Pre Data Access, the Post Data Access, or the On-fail exit points of an entity in a subclass:

- Specify the operation or operations of the entity superclass in the subclass.
- Enable the Pre Data Access, the Post Data Access, or the On-fail options, as appropriate, on the operations of the entity subclass.

In the example of the subclasses `EntityClass` and `EntitySubClass`, the subclass, `EntitySubClass`, inherits the key and details of the superclass. The same operations would be defined in both classes, for example, `insert`, `read`, and `modify`. In both these classes, the exit point options are enabled in the operations:

- 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 about exit points, see *Exit points* related link.

Related reference

Exit points on page 34
An exit point is a callback function that you write. It is executed at a predefined strategic point by the server.

1.29 Application customization

Store customizations separately from the original model. You can then upgrade the original model without overwriting customizations.

One of the more difficult aspects of customizing an application is managing upgrades to the original model. Any changes that are stored with the original model are overwritten when a newer version of the model is applied. To avoid overwriting customizations, you can store customizations separately from the original model.

The following features are available to help you customize an application:

- Extension classes.
- Overriding a domain definition.
- Subclass modeling.

For more information about extension classes, see the *Extension classes* related link.

For more information about overriding a domain definition, see the *Overriding a domain definition* related link.

For more information about subclass modeling, see the *Subclass modeling* related link.

Related reference

Extension classes on page 38
Use extension classes to specify options for a target class without modifying the meta-model definition of the target class.

Overriding a domain definition on page 27
Use the Server Development Environment (SDEJ) to override existing domain definitions without modifying the original domain definition. Use this feature in situations where the original domain definition is provided by a third party. Therefore, do not modify this feature locally.

Subclass modeling on page 113
Use subclassing for process, facade, entity, and wsinbound classes. Use subclassing to add new functionality or override existing functionality.

Related information

Notices

Privacy policy

Trademarks