DDS Data Local Reconstruction Layer (DLRL)

This OMG document accompanies the the RTF4 Recommendation and Report for this specification. The base document for the revised specification is formal/07-01-01.
This document is the result of applying to the adopted specification the issues resolved by the RTF4.

Contents

Contents i

Preface 1-iii

1. Overview 1-1

1.1 Introduction 1-1

1.2 Purpose 1-2

2. Data Local Reconstruction Layer (DLRL) 3-1

2.1 Platform Independent Model (PIM) 3-1

2.1.1 Overview and Design Rationale 3-1

2.1.2 DLRL Description 3-2

2.1.3 What Can Be Modeled with DLRL 3-2

2.1.4 Structural Mapping 3-6

2.1.5 Operational Mapping 3-13

2.1.6 Functional Mapping 3-13

2.2 OMG IDL Platform Specific Model (PSM) 3-45

2.2.1 Run-time Entities 3-45

2.2.2 Generation Process 3-62

2.2.3 Example 3-69

Compliance Points 1

Syntax for DLRL Queries and Filters 1

Preface

Object Management Group

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML

- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Suite 100
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Objective Interface Systems, Inc.
- Real-Time Innovations, Inc.
- THALES
- The Mitre Corporation
- University of Toronto

Overview 1

Contents

This chapter contains the following sections.

Section Title	Page
“Introduction”	1-1
“Purpose”	1-2

1.1 Introduction

This specification describes a high-level Data Local Reconstruction Layer (DLRL) interface to DDS that allows a simple integration of the DDS Service into the application layer.

DLRL uses typed interfaces. Typed interfaces, i.e., interfaces that take into account the actual data types, offer the following advantages:

- They are simpler to use: the programmer directly manipulates constructs that naturally represent the data.
- They are safer to use: verifications can be performed at compile time.
- They can be more efficient: the execution code can rely on the knowledge of the exact data type it has in advance, to e.g., pre-allocate resources.

It should be noted that the decision to use typed interfaces implies the need for a generation tool to translate type descriptions into appropriate interfaces and implementations that fill the gap between the typed interfaces and the generic middleware.

This specification is designed to allow a clear separation between the publish and the subscribe sides, so that an application process that only participates as a publisher can embed just what strictly relates to publication. Similarly, an application process that participates only as a subscriber can embed only what strictly relates to subscription.

1.2 Purpose

The DDS Specification offers an API that allows applications to access data in a logical Global Data Space. The DDS APIs provide direct access to this "shared data" in a uniform manner, meaning all applications share a common view of the data in terms of its Topic addressing and data-schemas.

However some applications require a local view of this data that is organized to fit the purpose and business logic of the application, which may be different for each individual process that accesses the Global Data.

This (DLRL) specification addresses this need by providing a convenient locally-defined Object API that abstracts the access to distributed information. The ***Data Local Reconstruction Layer (DLRL)*** automatically reconstructs data locally from the updates delivered by DDS allowing the application to access the data 'as if' it were local.

The combination of DLRL and DDS not only propagates the information to all interested subscribers but also updates a local copy of the information in the local format specified by each application.

Data Local Reconstruction Layer (DLRL) 2

Contents

This chapter contains the following sections.

Section Title	Page
“Platform Independent Model (PIM)”	2-1
“OMG IDL Platform Specific Model (PSM)”	2-45

2.1 Platform Independent Model (PIM)

DLRL stands for Data Local Reconstruction Layer. It is defined as a layer built on top of the Data-Distribution Service for Real-Time Systems (DDS).

2.1.1 Overview and Design Rationale

The purpose of DLRL is to provide more direct access to the exchanged data, seamlessly integrated with the native-language constructs. Object orientation has been selected for all the benefits it provides in software engineering.

As far as possible, DLRL is designed to allow the application developer to use the underlying DDS features. However, this may conflict with the main purpose of DLRL, which is ease of use and seamless integration into the application. Therefore, some DDS features may only be used through DDS and are not accessible from DLRL.

2.1.2 DLRL Description

With DLRL, the application developer will be able to:

- Describe classes of objects with their methods, data fields and relations.

- Attach some of those data fields to DCPS entities.
- Manipulate those objects (i.e., create, read, write, delete) using the native language constructs that will, behind the scenes, activate the attached DCPS entities in the appropriate way.
- Have those objects managed in a cache of objects, ensuring that all the references that point to a given object actually point to the same language cell.

This specification explains the following:

- Which object-oriented constructs can be used to define DLRL objects.
- Which functions are applicable to those objects (e.g., create, delete, etc.).
- The different levels of mapping between the two layers:
  - structural mapping (i.e., relations between DLRL objects and DCPS data).
  - operational mapping (i.e., mapping of the DLRL objects to the DCPS entities (***Publisher***, ***DataWriter***, etc.) including QoS settings, combined subscriptions.
  - functional mapping (i.e., relations between the DLRL functions (mainly access to the DLRL objects) and the DCPS functions (write/publish).

2.1.3 What Can Be Modeled with DLRL

2.1.3.1 DLRL objects

DLRL allows an application to describe objects with:

- methods;
- attributes that can be:
  - local (i.e., that do not participate in the data distribution) or,
  - shared (i.e., that participate in the data distribution process and are thus attached to DCPS entities).

Only shared attributes are of concern to the Data Distribution Service; therefore, the remaining part of this document will only deal with these properties.

A DLRL object has at least one shared attribute. Shared attributes are typed[1] and can be either mono-valued or multi-valued:

- Mono-valued:
  - of a simple type:
    - basic-type (long, short, char, string, etc.)
    - enumeration-type
    - simple structure[2]
  - reference to a DLRL object.

1. At the PIM level, we describe the minimum set that is required to describe shared attributes. This does not prevent a specific PSM from extending this minimum set, in case this make sense and does not affect the ability of this layer to be implemented on top of DCPS.

For these mono-valued attributes, type enforcement is as follows:

- Strict type equality for simple types.
- Equality based on inclusion for reference to a DLRL object (i.e., a reference to a derived object can be placed in a reference to a base object).

- Multi-valued (collection-based):
  - two collection basis of homogeneously-typed items:
    - a list (ordered with index)
    - a map (access by key).
    - a set (not ordered).

Type enforcement for collection elements is as follows:

- Strict type equality for simple types.
- Equality based on type inclusion for references to DLRL objects (i.e., a reference to a derived object can be placed in a collection typed for base objects).

DLRL will manage DLRL objects in a cache (i.e., two different references to the same object – an object with the same identity – will actually point to the same memory location).

Object identity is given by an ***oid*** (object ID) part of any DLRL object.

2.1.3.2 Relations among DLRL objects

Relations between DLRL objects are of two kinds:

- Inheritance that organizes the DLRL classes.
- Associations that organize the DLRL instances.

2.1.3.2.1 Inheritance

Single inheritance is allowed between DLRL objects.

Any object inheriting from a DLRL object is itself a DLRL object.

ObjectRoot is the ultimate root for all DLRL objects.

DLRL objects can, in addition, inherit from any number of native language objects.

2.1.3.2.2 Associations

Supported association ends are either *to-1* or *to-many*. In the following, an association end is named a *relation*:

- to-1 relation is featured by a mono-valued attribute (reference to the target object).
- to-many relation is featured by a multi-valued attribute (collection of references to the target objects).

---

2. For instance, structures that can be mapped inside one DCPS data.

Supported relations are:

- Plain use-relations (no impact on the object life-cycle).
- Compositions (constituent object lifecycle follows the compound object's one).

Couples of relations can be managed consistently (one being the *inverse* of the other), to make a real association (in the UML sense):

- One plain relation can inverse another plain relation, providing that the types match: can make 1-1, 1-n, n-m.
- One composition relation can only inverse a to-1 relation to the compound object: can make 1-1 or 1-n.

Note – Embedded structures are restricted to the ones that can be mapped simply at the DCPS level. For more complex ones, component objects (i.e., objects linked by a composition relation) may be used.

2.1.3.3 Metamodel

The following figure represents the DLRL metamodel, i.e., all the constructs that can be used to describe the 'shared' part of a DLRL model. This metamodel is given for explanation purpose. This specification does not require that it is implemented as such.

Note that two objects that will be part of a DLRL model (namely ***ObjectRoot*** that is the root for all the DLRL classes as well as ***ObjectHome*** that is the class responsible for creating and managing all DLRL objects of a given class) are featured to show the conceptual relations between the metamodel and the model. They appear in grey on the schema.

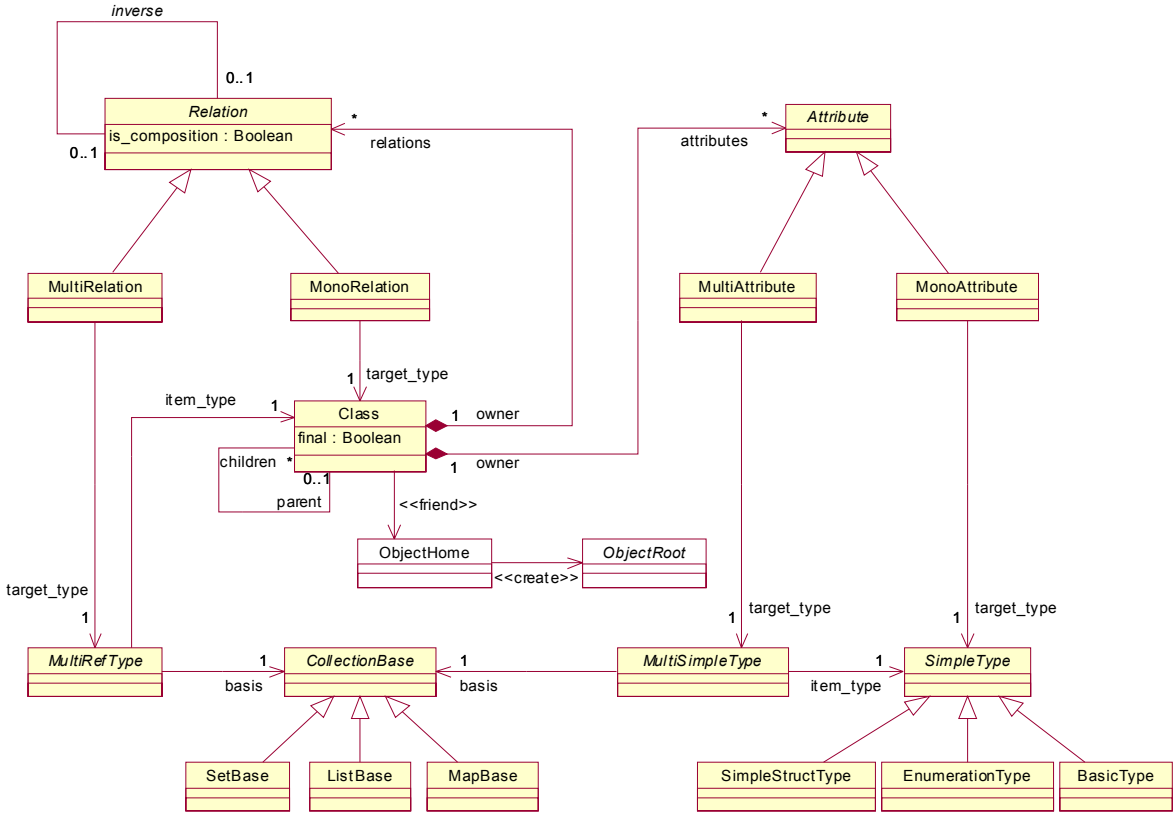


Figure 2-1 DLRL Metamodel

Instances of ***BasicType*** are:

- long
- short
- char
- octet
- real
- double
- string
- sequence of any of the above

Instances of ***EnumerationType*** are all the enumerations.

Instances of ***SimpleStructType*** are all the structures that can be mapped in a single DCPS data.

2.1.4 Structural Mapping

2.1.4.1 Design Principles

The mapping should not impose unnecessary duplication of data items.

The mapping should not prevent an implementation from being efficient. Therefore, adding information in DCPS data to help DLRL internal management is allowed.

The mapping should be as flexible as possible. It is therefore specified on an attribute basis (that means that *any* attribute, even a simple one, can be located in a DCPS data structure that is separate from the main one; i.e., the DCPS data structure associated with the DLRL class)[3].

This flexibility is highly desirable to meet specific requirements (e.g., to reuse an existing DCPS description). However, there are cases when this type of flexibility is not needed and leads to extra descriptions that could (and should) be avoided. For these cases, a default mapping is also defined.

2.1.4.2 Mapping rules

Recall that DCPS data can be seen as tables (***Topic***) whose rows correspond to instances identified by their key value and whose columns (fields) correspond to data fields. Each cell contains the value of a given field for a given instance and the key value is the concatenation of the values of all the fields that make the key definition (itself attached to the ***Topic***).

Structural mapping is thus very close to Object to Relational mapping in database management.

Generally speaking, there is some flexibility in designing the DCPS model that can be used to map a DLRL model. Nevertheless, there are cases where the underlying DCPS model exists with no provision for storing the object references and no way to modify them. In that case however, the DCPS topics contain fields (the keys) that allow the unique identification of instances. With some restrictions concerning inheritance, these models can also be mapped back into DLRL models. Section 2.1.4.5, “Mapping when DCPS Model is Fixed,” on page 2-11 is specifically dedicated to that issue.

The mapping rules when some flexibility is allowed in DCPS model are as follows.

2.1.4.2.1 Mapping of Classes

Each DLRL class is associated with at least one DCPS table, which is considered as the ‘main’ table. A DLRL object is considered to exist if it has a corresponding row in this table. This table contains at least the fields needed to store a reference to that object (see below).

3. This is needed to efficiently manage inheritance. Therefore extending it to any attribute is not costly.

To facilitate DLRL management and save memory space, it is generally desirable that a derived class has the same main table as its parent concrete class (if any)[4], with the attributes that are specific to the derived class in an extension table. For example, this allows the application to load all the instances of a given class (including its derivations) in a single operation.

2.1.4.2.2 Mapping of an Object Reference

To reference an object, there must be a way to designate it unambiguously and a way to retrieve the exact class of that object (this last point is needed when the object has to be locally created based on received information).

Therefore, to reference an object, the following must be stored:

- A string that allows retrieval of the exact class (e.g., name class, or more precisely a public name that identifies the class unambiguously).
- A number that identifies the object inside this class[5] (***oid***).

The combination of these two pieces of information is called *full* ***oid***.

There are cases where the indication of the class is not needed, for it can be deduced from the knowledge embedded in the mapping. A class name is needed when:

- Several classes share the same main table.
- Several classes are targets for the same relation (in other words, when the target type of a relation is a class that has derived classes).

2.1.4.2.3 Mapping of Attributes and Relations

Mono-valued attributes and relations are mapped to one (or several) cell(s)[6] in a single row whose key is the means to unambiguously reference the DLRL object (i.e., its ***oid*** or its full ***oid***, depending on the ***owner*** class characteristics as indicated in the previous section):

- simple basic attributes -> one cell of corresponding DCPS type;
- enumeration -> one cell of type integer[7] (default behavior) or string;
- simple structures -> as many cells as needed to hold the structure;
- reference to another DLRL object (i.e., relation) -> as many cells as needed to reference unambiguously the referenced object (i.e., its ***oid***, or its full ***oid*** as indicated in the previous section).

---

4. Excluding, of course, the abstract ObjectRoot (otherwise all the objects will be located in a single table).
5. Note that, in case several parts are creating objects at the same time, there should be a means to guarantee that there is no confusion (e.g., by means of two sub-fields, one to designate the author and one for a sequence number). This is left to the implementation.
6. Depending of the type of the value.

Multi-valued attributes are mapped to one (or several) cell(s) in a set of rows (as many as there are items in the collection), whose key is the means to unambiguously designate the DLRL object (i.e., ***oid*** or full ***oid***) plus an index in the collection.

- For each item, there is one row that contains the following, based on the type of attribute:
  - simple basic type -> one cell of the corresponding DCPS type;
  - enumeration -> one cell of type integer or string;
  - simple structures -> as many cells as needed to hold the structure;
  - reference to another DLRL object -> as many cells as needed to reference unambiguously the referenced object (i.e., its oid, or its full ***oid*** as indicated in the previous section).
- The key for that row is the means to designate the owner's object (i.e., its ***oid*** or full ***oid***) + an index, which is:
  - An integer if the collection basis is a list (to hold the rank of the item in the list).
  - A string or an integer[8] if the collection basis is a map (to hold the access key of the item in the map).

2.1.4.3 Default Mapping

The following mapping rules will be applied by default. This default mapping is overwritten by any mapping information provided by the application developer.

- Main table
  - Name of the DCPS Topic is the DLRL class name.
  - Name of the ***oid*** fields are:
    - "class"
    - "oid"
- All the mono-valued attributes of an object are located in that main table
  - name of the DCPS Topic is thus DLRL class name;
  - name of the DCPS fields:
    - name of the DLRL attribute, if only one field is required;
    - name of the DLRL attribute, concatenated with the name of each sub-field, with '.' as separator, otherwise.
- For each multi-valued attribute, a specific DCPS table is allocated
  - name of the DCPS Topic is the DLRL class name concatenated with the DLRL attribute name, with '.' as separator;
  - name of the DCPS fields:

---

7. In the PIM, the type 'integer' has been chosen each time a whole number is needed. In the PSM, however, a more suitable representation for such numbers (long, short...) will be chosen.
8. String-keyed maps are desired for their openness; however, integer-keyed maps are more suitable when access performance is desired.

- same as above for the value part and the OID part
- "index" for the extra key field

- Inheritance support by means of extension tables gathering all the mono-valued added attributes:
  - this choice is the better as far as memory is concerned;
  - it is made possible once it is admitted that all the attributes of a given class are not located in a single table.

2.1.4.4 Metamodel with Mapping Information

Figure 2-2 represents the DLRL metamodel with the information that is needed to indicate the structural mapping.

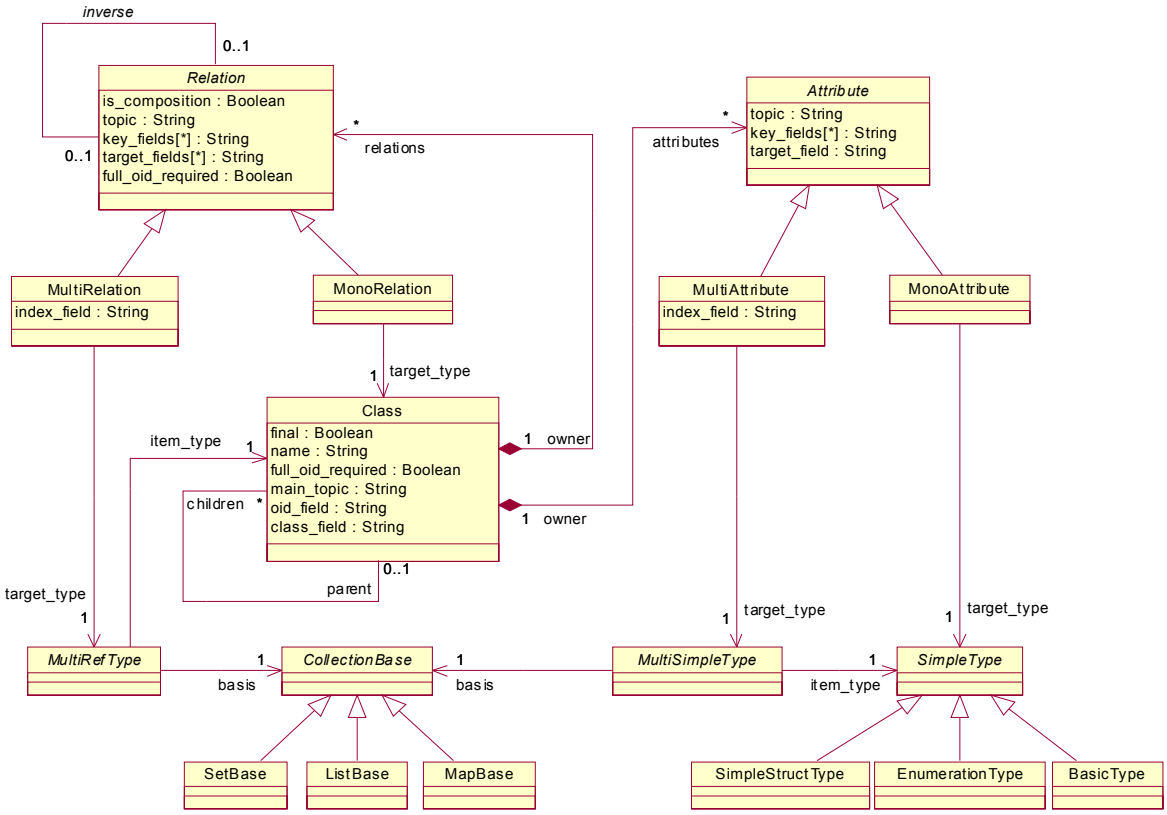


Figure 2-2 DLRL Model with Structural Mapping Information

The three constructs that need added information related to the structural mapping are ***Class***, ***Attribute***, and ***Relation***.

2.1.4.4.1 Class

The related fields have the following meaning:

- ***main_topic*** is the name of the main topic for this class. Any DLRL instance of this ***Class*** is represented by a row in this ***topic***[9].

- ***oid_field*** is the name of the field meant to store the ***oid*** of the DLRL object.
- ***class_field*** is the name of the field meant to store the ***name*** of the ***Class***.
- ***full_oid_required*** indicates whether the class ***name*** should be the first part of the actual key; the actual key will be made of:
  - (***class_field***, ***oid_field***) if it is true.
  - (***oid_field***) if it is false.
- ***final*** indicates whether or not the class can be extended.

2.1.4.4.2 MonoAttribute

The related fields have the following meaning:

- ***topic*** is the name of the table where the related value is located. It may be the same as the ***owner Class::main_topic***.
- ***target_field*** is the field that contains the actual value for the attribute.
- ***key_fields*** is the name of the fields that make the key in this topic (1 or 2 depending on the ***Class*** definition).

2.1.4.4.3 MultiAttribute

The related fields have the following meaning:

- ***topic*** is the name of the table where the related value is located. It cannot be the same as the ***owner Class::topic***.
- ***target_field*** is the field that contains the actual values for the attribute.
- ***key_fields*** is the name of the fields that make the object part of the key in this topic (1 or 2 depending on the ***owner Class*** definition).
- ***index_field*** is the name of the item part of the key in this ***topic*** (string or integer depending on the collection type)[10].

2.1.4.4.4 MonoRelation

The related fields have the following meaning:

- ***topic*** is the name of the table where the related value is located. It may be the same as the ***owner Class::topic***.
- ***target_fields*** are the fields that contain the actual value for the attribute (i.e., what identifies the target object). It is made of 1 or 2 fields according to the ***full_oid_required*** value).

---

9. It may have attributes in other topics as well.
10. In other words, all the rows that have the same value for the key_fields constitute the contents of the collection; each individual item in the collection is pointed by (key_fields, index_field).

- ***key_fields*** is the name of the fields that make the key in this topic (1 or 2 depending on the ***owner Class*** definition).
- ***full_oid_required*** indicates whether that relation needs the full ***oid*** to designate target objects.
- ***is_composition*** indicates if it is a mono- or multi-relation.

2.1.4.4.5 MultiRelation

The related fields have the following meaning:

- ***topic*** is the name of the table where the related value is located. It cannot be the same as the ***owner Class::topic***.
- ***target_field***s are the fields that contain the actual values for the attribute (i.e., what identify the target objects). It is made of 1 or 2 fields according to the ***full_oid_required*** value).
- ***key_fields*** is the name of the fields that make the object part of the key in this ***topic*** (1 or 2 depending on the ***owner Class*** definition).
- ***index_field*** is the name of the item part of the key in this topic (string or integer depending on the collection type).
- ***full_oid_required*** indicates whether that relation needs the full ***oid*** to designate target objects.
- ***is_composition*** indicates if it is a mono- or multi-relation.

2.1.4.5 Mapping when DCPS Model is Fixed

In some occasions, it is desirable to map an existing DCPS model to the DLRL. It is even desirable to mix, in the same system, participants that act at DCPS level with others that act at the DLRL level. The DLRL, by not imposing the same object model to be shared among all participants, is even designed to allow this last feature.

In this case, it is possible to use the topic keys to identify the objects, but not to store the object references directly. Therefore, the DLRL implementation must indicate the topic fields that are used to store the keys so that, behind the scenes, it can manage the association keys to/from oid and perform the needed indirection.

Because the object model remains local, this is feasible even if supporting inheritance between the applicative classes (beyond the primary inheritance between an applicative class and ***ObjectRoot***) may be tricky. However an exiting DCPS model by construction is unlikely to rely heavily on inheritance between its ‘classes.’ Therefore such a mapping is supported.

2.1.4.6 How is this Mapping Indicated?

There should be two orthogonal descriptions:

- The object model itself, i.e.,
  - the full object model,

  - indications of the part that is to be made shared.
- The mapping itself.

In case we were targeting only languages where metaclasses are fully supported, this information could be provided by the application developer by instantiating the above mentioned constructs. As this is not the case, we propose the following approach, as described on Figure 2-3.

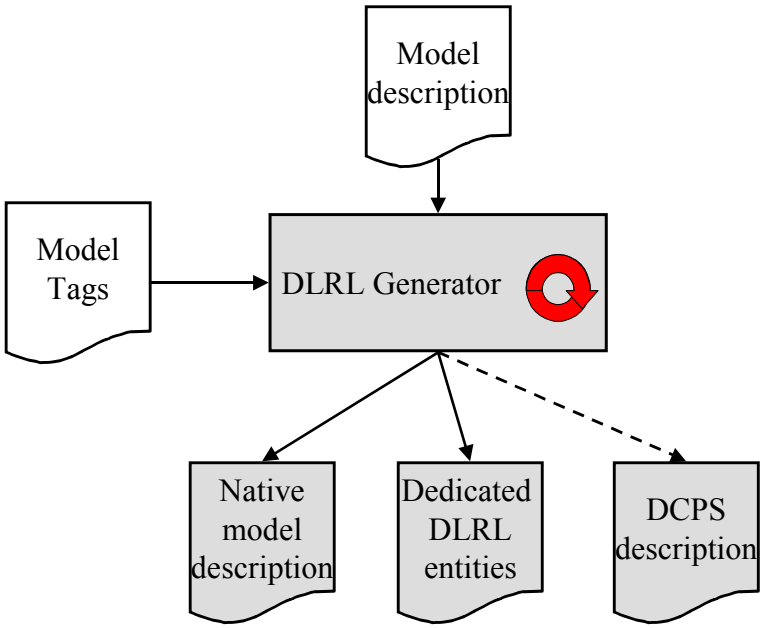


Figure 2-3 DLRL Generation Process

Based on the model description and tags that enhance the description, the tool will generate:

- The native model definition (i.e., the application classes as they will be usable by the application developer).
- The dedicated DLRL entities (i.e., the helper classes to consistently use the former ones and form the DLRL run-time).
- On demand, the corresponding DCPS description.

The syntax of those descriptions is dependant on the underlying platform. One syntax is proposed with the OMG IDL PSM in Section 2.2, “OMG IDL Platform Specific Model (PSM),” on page 2-45.

2.1.5 Operational Mapping

2.1.5.1 Attachment to DCPS Entities

A DLRL class is associated with several DCPS ***Topic***, each of which is accessible via a DCPS ***DataWriter*** (write access) and/or a DCPS ***DataReader*** (read access). All the ***DataWriter***/***DataReader*** objects that are used by a DLRL object are to be attached to a single ***Publisher***/***Subscriber*** in order to consistently manage the object contents.

DLRL classes are linked to other DLRL classes by means of relations. In order for these relations to be managed consistently (e.g., when a relation is set to a newly created object, set up of the relation and the object creation are simultaneously performed), the whole graph has to be attached to the same ***Publisher***/***Subscriber***.

Therefore, DLRL has attached a ***Publisher*** and/or a ***Subscriber*** to the notion of a ***Cache*** object, which manages all the objects, thereby making a consistent set of related objects. The use of those DCPS entities is thus totally transparent to the application developer.

2.1.5.2 Creation of DCPS Entities

Operations are provided at the DLRL level to create and activate all the DCPS entities that are needed for managing all the instances of DLRL classes attached to a ***Cache***, for publication and/or for subscription.

Note – Activating the related DCPS entities for subscription (namely the *Subscriber* and its attached *DataReader* objects) corresponds to actually performing the subscriptions.

2.1.5.3 Setting of QoS

QoS must be attached to each DCPS entity (***Publisher***/***Subscriber***, ***Topic***/***DataWriter***/***DataReader***). This can be done between the creation and activation of these entities.

Putting the same QoS on all the DCPS entities that are used for a graph of objects (or even for a single object) is not very sensible. In return, it is likely that one object will present different attributes with different QoS requirements (i.e., some parts of the object need to be PERSISTENT, others are VOLATILE). Therefore, DLRL does not offer a specific means to set QoS, but it does offer a means to retrieve the DCPS entities that are attached to the DLRL entities, so that the application developer can set QoS if needed.

2.1.6 Functional Mapping

Functional mapping is the translation of the DLRL functions to DCPS functions. It obviously depends firstly on the DLRL operation modes (i.e., the way the applications may use the DLRL entities).

2.1.6.1 DLRL Requested Functions

2.1.6.1.1 Publishing Application

Once the publishing DCPS infrastructure is set, publishing applications need to repeatedly:

- create objects,
- modify them,
- possibly destroy them,
- request publication of the performed changes (creations, modifications, destructions).

Even if an object is not changeable by several threads at the same time, there is a need to manage concurrent threads of modifications in a consistent manner.

2.1.6.1.2 Subscribing Application

Once the subscribing DCPS infrastructure is set, subscribing applications need to:

- load objects (i.e., make subscribed DCPS data, DLRL objects);
- read their attributes and/or relations;
- possibly use the relations to navigate among the objects;
- be made aware of changes to the objects that are there, or the arrival of new objects.

The application needs to be presented with a consistent view of a set of objects.

2.1.6.1.2.1 Implicit versus Explicit Subscriptions

The first important question is whether the loading of objects happens in the scope of the known subscriptions (explicit subscriptions) or whether it may extend them, especially when navigating to another object by means of a relation (implicit subscriptions). The choice has been to keep the DLRL set of objects inside the boundary of the known subscriptions[11], for the following reasons:

- In the use cases we have, implicit subscriptions are not needed.
- Implicit subscriptions would cause the following issues, which are almost impossible to solve while maintaining a high level of decoupling between DCPS and DLRL:
  - Structural mapping - to which DCPS data does the new object definition correspond?
  - Operational mapping - in particular, which QoS has to be associated to the related DCPS entities?

11. That means that no subscription will be made “on the fly” to reach an object that is an instance of a class for which no subscription has been made.

- Implicit subscriptions would make it difficult for the application to master its set of objects.

If a relation points towards an object for which no subscription exists, navigating through that relation will raise an error (NotFound).

2.1.6.1.2.2 Cache Management

The second important question is how the cache of objects is updated with incoming information. This can be done:

- upon application requests,
- fully transparently.

DLRL general principle is to update the cache of objects transparently with incoming updates. However, means are given to the application to turn on/off this feature when needed. In addition, copies of objects can be requested in order to navigate into a consistent set of object values when updates continue to be applied on the originals (see ***CacheAccess*** objects for more details).

2.1.6.1.2.3 User Interaction

Another important question is how the application is made aware of changes on the objects it has. A listener is a convenient pattern for that purpose. The question is, however, the granularity it gets:

- It is useful to reflect several incoming updates ‘as a whole.’
- For an object modification, it is useful to indicate which are the modified attributes.

2.1.6.1.3 Publishing and Subscribing Applications

Most of DLRL publishing applications will also be subscribing ones. There is thus a strong need to support this nicely. In particular, it means that the application should be able to control the mix of incoming updates and of modifications it performs.

2.1.6.2 DLRL Entities

Figure 2-4 describes all the DLRL entities that support the DLRL operations at run-time. Note that most of them are actually roots for generated classes depending on the DLRL classes (they are indicated in *italics*); the list of classes that are generated for an application-defined class named ***Foo*** is given in Section 2.1.6.6, “Generated Classes,” on page 2-44.

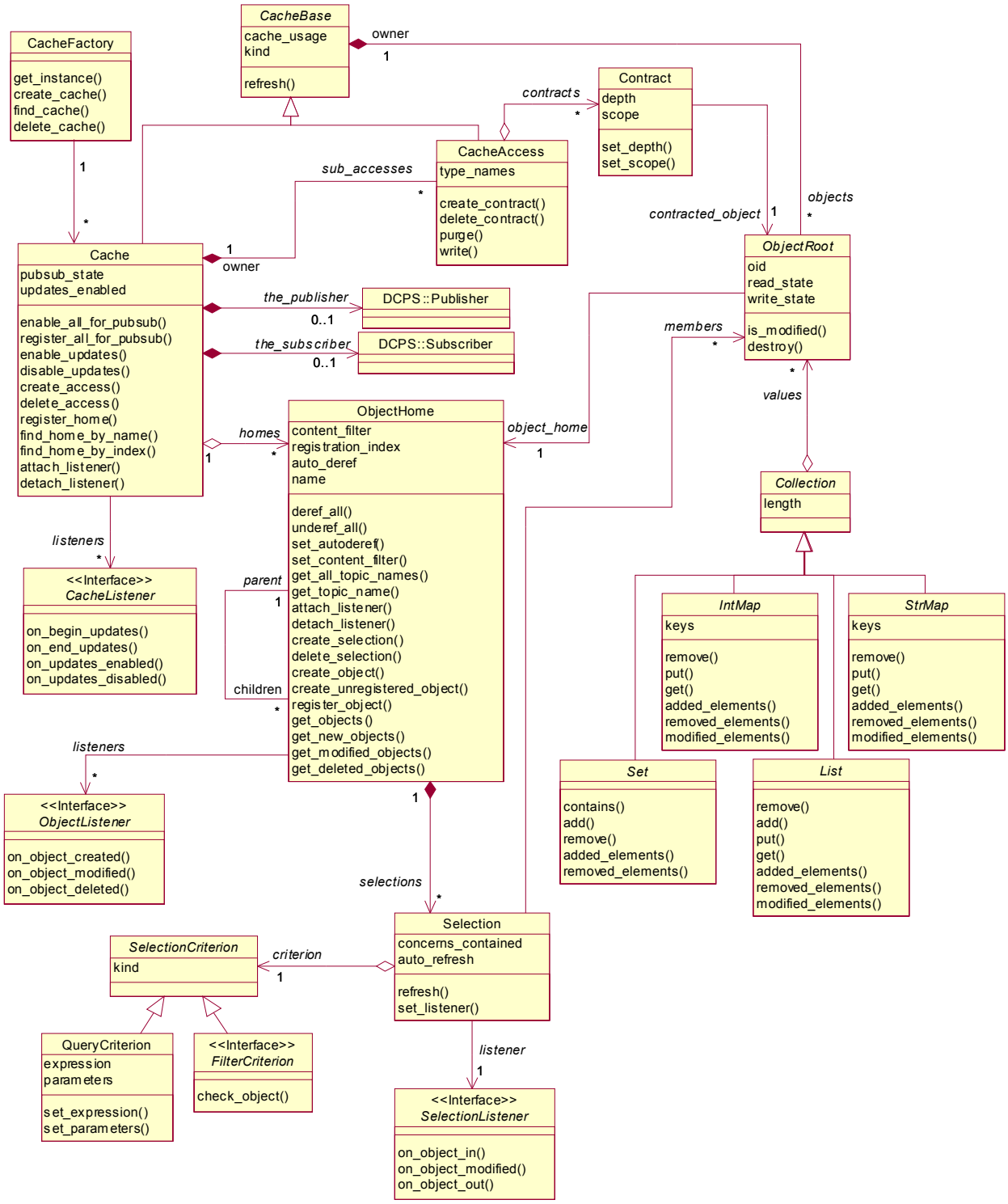


Figure 2-4 DLRL entities

The DLRL entities are:

CacheFactory	Class whose unique instance allows the creation of ***Cache*** objects.
CacheBase	Base class for all Cache types.
Cache	Class whose instance represents a set of objects that are locally available. Objects within a ***Cache*** can be read directly; however to be modified, they need to be attached first to a ***CacheAccess***. Several ***Cache*** objects may be created but in this case, they must be fully isolated: • A ***Publisher*** can only be attached to one ***Cache***. • A ***Subscriber*** can only be attached to one ***Cache***. • Only DLRL objects belonging to one ***Cache*** can be put in relation.
CacheAccess	Class that encapsulates the access to a set of objects. It offers methods to refresh and write objects attached to it; ***CacheAccess*** objects can be created in read mode, in order to provide a consistent access to a subset of the ***Cache*** without blocking the incoming updates or in write mode in order to provide support for concurrent modifications/updates threads.
CacheListener	Interface to be implemented by the application to be made aware of the arrival of incoming updates on the cache of objects.
Contract	Class that represents a contract between a ***CacheAccess*** and a ***Cache*** that defines which objects will be cloned from the ***Cache*** into the ***CacheAccess*** when the latter is refreshed.
ObjectHome	Class whose instances act as representative for all the local instances of a given application-defined class.
ObjectListener	Interface to be implemented by the application to be made aware of incoming updates on the objects belonging to one peculiar ***ObjectHome***.
Selection	Class whose instances act as representatives of a given subset of objects. The subset is defined by an expression attached to the selection.
SelectionCriterion	Class whose instances act as filter for Selection objects. When a Selection is created, it must be given an ***SelectionCriterion***.
FilterCriterion	Specialization of ***SelectionCriterion*** that performs a filter based on user-defined filter algorithm.

QueryCriterion	Specialization of ***SelectionCriterion*** that performs a filter based on a query expression.
SelectionListener	Interface to be implemented by the application to be made aware on updates made on objects belonging to that selection.
ObjectRoot	Abstract root class for all the application-defined classes.
Collection	Abstract root for all the collections of objects as well as of values.
List	Abstract root for all the lists of objects as well as of values.
Set	Abstract root for all the sets of objects as well as of values.
StrMap	Abstract root for all the maps of objects as well as of values, with string key management.
IntMap	Abstract root for all the maps of objects as well as of values, with integer key management.

The DLRL API may raise Exceptions under certain conditions. What follows is an extensive list of all possible Exceptions and the conditions in which they will be raised:

- "***DCPSError***: if an unexpected error occured in the DCPS
- "***BadHomeDefinition***: if a registered ***ObjectHome*** has dependencies to other, unregistered ***ObjectHomes***.
- "***NotFound***: if a reference is encountered to an object that has not (yet) been received by the DCPS.
- "***AlreadyExisting***: if a new object is created using an identify that is already in use by another object.
- "***AlreadyDeleted*** - if an operation is invoked on an object that has already been deleted.
- "***PreconditionNotMet*** - if a precondition for this operation has not (yet) been met.
- "***NoSuchElement*** - if an attempt is made to retrieve a non-existing element from a Collection.
- "***SQLError*** - if an SQL expression has bad syntax, addresses non-existing fields or is not consistent with its parameters.

Each exception contains a string attribute named '***message***', that gives a more precise explanation of the reason for the exception.

2.1.6.3 Details on DLRL Entities

The following sections describe each DLRL entity one by one. For each entity a table summarizes the public attributes and/or methods provided.

It should be noted that, as far as the return value of a method is concerned, only the functional values are indicated. Errors are not considered here. At PSM level, a consistent scheme for error returning will be added.

When a parameter or a return value is stated as ‘undefined,’ that means that the operation is actually part of an abstract class, which will be derived to give concrete classes with typed operations.

2.1.6.3.1 CacheFactory

The unique instance of this class allows the creation of Cache objects.

CacheFactory		
no attributes		
operations		
(static) get_instance		CacheFactory
create_cache		Cache
	cache_usage	CacheUsage
	description	CacheDescription
find_cache_by_name		Cache
	name	CacheName
delete_cache		void
	a_cache	Cache

This class offers methods:

- To retrieve the ***CacheFactory*** singleton. The operation is idempotent, that is, it can be called multiple times without side-effects and it will return the same ***CacheFactory*** instance. The ***get_instance*** operation is a static operation implemented using the syntax of the native language and can therefore not be expressed in the IDL PSM
- To create a Cache ***(create_cache)***.
  This method takes as a parameter ***cache_usage***, which indicates the future usage of the ***Cache*** (namely WRITE_ONLY—no subscription, READ_ONLY—no publication, or READ_WRITE—both modes) and a description of the ***Cache*** (at a minimum, this ***CacheDescription*** gathers the concerned ***DomainParticipant*** as well as a ***name*** allocated to the ***Cache***). Depending on the ***cache_usage*** a ***Publisher***, a ***Subscriber***, or both will be created for the unique usage of the ***Cache***. These two objects will be attached to the passed ***DomainParticipant***.
- To retrieve a ***Cache*** based on the name given in the ***CacheDescription*** (***find_cache_by_name***). If the specified name does not identify an existing ***Cache***, a NULL is returned.
- To delete a ***Cache*** (***delete_cache***). This operation releases all the resources allocated to the ***Cache***.

2.1.6.3.2 CacheBase

CacheBase is the base class for all Cache classes. It contains the common functionality that supports ***Cache*** and ***CacheAccess***.

CacheBase		
attributes		
cache_usage	CacheUsage	
objects	ObjectRoot[]	
kind	CacheKind	
operations		
refresh		void

The public attributes give:

- "The ***cache_usage*** indicates whether the cache is intended to support write operations (WRITE_ONLY or READ_WRITE) or not (READ_ONLY). This attribute is given at creation time and cannot be changed afterwards.
- "A list of (untyped) objects that are contained in this ***CacheBase***. To obtain objects by type, see the ***get_objects*** method in the typed ***ObjectHome***.

The kind describes whether a ***CacheBase*** instance represents a ***Cache*** or a ***CacheAccess***.It offers methods to:

- "Refresh the contents of the Cache with respect to its origins (DCPS in case of a main ***Cache***, ***Cache*** in case of a ***CacheAccess***).

2.1.6.3.3 CacheAccess

CacheAccess is a class that represents a way to globally manipulate DLRL objects in isolation.

CacheAccess : CacheBase	
attributes	
owner	Cache
contracts	Contract[]
type_names	string[]
operations	

write		void
purge		void
create_contract		Contract
	object	ObjectRoot
	scope	ObjectScope
	depth	long
delete_contract		void
	a_contract	Contract

A ***CacheAccess*** only belongs to one ***Cache*** (***owner***)—the one that creates it.

The public attributes give:

- The owner of the ***Cache*** (***owner***)
- The contracted objects (***contracts***). This is the list of all Contracts that are attached to this ***CacheAccess***.
- "A list of names that represents the types for which the ***CacheAccess*** contains at least one object (***type_names***).

The ***CacheAccess*** offers methods to:

- Write objects (***write***). If the ***CacheAccess::cache_usage*** allows write operation, those objects can be modified and/or new objects created for that access and eventually all the performed modifications written for publications.
- Detach all contracts (including the contracted DLRL Objects themselves) from the ***CacheAccess*** (***purge***)."
- "Create a ***Contract*** (***create_contract***). This method defines a contract that covers the specified object with all the objects in its specified scope. When a ***CacheAccess*** is refreshed, all contracted objects will be cloned into it. The contracted object must be located in the ***Cache*** that owns the ***CacheAccess***. If this is not the case, a ***PreconditionNotMet*** is raised.
- "Delete a ***Contract*** (***delete_contract***). This method deletes a contract from the ***CacheAccess***. When the ***CacheAccess*** is refreshed, the objects covered by the specified contract will no longer appear in the ***CacheAccess*** (unless also covered in another ***Contract***). The specified ***Contract*** must be attached to this ***CacheAccess***, otherwise a ***PreconditionNotMet*** is raised

See Section 2.1.6.5, “Cache Accesses Management,” on page 2-43 for a description of typical uses of cache accesses.

2.1.6.3.4 Cache

An instance of this class gathers a set of objects that are managed, published and/or subscribed consistently.

Cache : CacheBase		
attributes		
pubsub_state	DCPSState	
updates_enabled	boolean	
sub_accesses	CacheAccess []	
homes	ObjectHome []	
listeners	CacheListener []	
the_publisher	DDS::Publisher	
the_subscriber	DDS::Subscriber	
operations		
register_home		integer
	a_home	ObjectHome
find_home_by_name		ObjectHome
	class_name	string
find_home_by_index		ObjectHome
	index	integer
register_all_for_pubsub		void
enable_all_for_pubsub		void
attach_listener		void
	listener	CacheListener
detach_listener		void
	listener	CacheListener
enable_updates		void
disable_updates		void
load		void
create_access		CacheAccess
	purpose	CacheUsage
delete_access		void
	access	CacheAccess
lock		void
	to_in_milliseconds	integer
unlock		void

The public attributes give:

- the state of the cache with respect to the underlying Pub/Sub infrastructure (***pubsub_state***), as well as the related ***Publisher*** (*the_publisher*) and ***Subscriber*** (*the_subscriber*).

- the state of the cache with respect to incoming updates (***updates_enabled***). This state is modifiable by the applications (see ***enable_updates***, ***disable_updates***) in order to support applications that are both publishing and subscribing.
- the attached ***CacheAccess*** (*sub_accesses*).
- the attached ***ObjectHome*** (*homes*).
- the attached ***CacheListener*** (*listeners*).

It offers methods to:

- register an ***ObjectHome*** (***register_home***). This method returns the index under which the ***ObjectHome*** is registered by the ***Cache***. A number of preconditions must be satisfied when invoking the ***register_home*** method: the ***Cache*** must have a ***pubsub_state*** set to INITIAL, the specified ***ObjectHome*** may noy yet be registered before (either to this ***Cache*** or to another ***Cache***), and no other instance of the same class as the specified ***ObjectHome*** may already have been registered to this ***Cache***. If these preconditions are not satisfied, a ***PreconditionNotMet*** is raised
- retrieve an already registered ***ObjectHome*** based on its name (***find_home_by_name***) or based on its index of registration (***find_home_by_index***). If no registered home can be found that satisfies the specified name or index, a NULL is returned.
- register all known ***ObjectHome*** to the Pub/Sub level (***register_all_for_pubsub***), i.e., create all the needed DCPS entities; registration is performed for publication, for subscription or for both according to the ***cache_usage***. At this stage, it is the responsibility of the service to ensure that all the object homes are properly linked and set up: that means in particular that all must have been registered before. When an ***ObjectHome*** still refers to another ***ObjectHome*** that has not yet been registered, a ***BadHomeDefinition*** is raised. A number of preconditions must also be satisfied before invoking the ***register_all_for_pubsub*** method: at least one ***ObjectHome*** needs to have been registered, and the ***pubsub_state*** may not yet be ENABLED. If these preconditions are not satisfied, a ***PreconditionNotMet*** will be raised. Invoking the ***register_all_for_pub_sub*** on a REGISTERED ***pubsub_state*** will be considered a no-op.
- enable the derived Pub/Sub infrastructure (***enable_all_for_pubsub***). QoS setting can be performed between those two operations. One precondition must be satisfied before invoking the ***enable_all_for_pub_sub*** method: the ***pubsub_state*** must already have been set to REGISTERED before. A ***PreconditionNotMet*** Exception is thrown otherwise. Invoking the ***enable_all_for_pub_sub*** method on an ENABLED ***pubsub_state*** will be considered a no-op.
- attach/detach a ***CacheListener*** (***attach_listener***, ***detach_listener***).
- enable/disable incoming updates (***enable_update***s, ***disable_updates***):
  - ***disable_updates*** causes incoming but not yet applied updates to be registered for further application. If it is called in the middle of a set of updates (see ***Listener*** operations), the ***Listener*** will receive ***end_updates*** with a parameter that indicates that the updates have been interrupted.
  - ***enable_updates*** causes the registered (and thus not applied) updates to be taken into account, and thus to trigger the attached ***Listener***, if any.

- explicitly request taking into account the waiting incoming updates (***load***). In case ***updates_enabled*** is TRUE, the load operation does nothing because the updates are taken into account on the fly; in case ***updates_enabled*** is FALSE, the ***load*** operation 'takes' all the waiting incoming updates and applies them in the ***Cache***. The ***load*** operation does not trigger any listener (while automatic taking into account of the updates does - see Section 2.1.6.4, “Listeners Activation,” on page 2-41 for more details on listener activation) and may therefore be useful in particular for global initialization of the ***Cache***.
- create new ***CacheAccess*** objects dedicated to a given purpose (***create_access***). This method allows the application to create sub-accesses and takes as a parameter the purpose of that sub-access, namely:
  - write allowed (WRITE_ONLY or READ_WRITE[12]) – to isolate a thread of modifications.
  - write forbidden (READ_ONLY) – to take a consistent view of a set of objects and isolate it from incoming updates.

The purpose of the ***CacheAccess*** must be compatible with the usage mode of the ***Cache***: only a ***Cache*** that is write-enabled can create a ***CacheAccess*** that allows writing. Violating this rule will raise a ***PreconditionNotMet***:

- delete sub-accesses (***delete_access***). Deleting a ***CacheAccess*** will purge all its contents. Deleting a ***CacheAccess*** that is not created by this ***Cache*** will raise a ***PreconditionNotMet***.
- transform an ***ObjectReference*** to the corresponding ***ObjectRoot***. This operation can return the already instantiated ***ObjectRoot*** or create one if not already done. These ***ObjectRoot*** are not modifiable (modifications are only allowed on cloned objects attached to a ***CacheAccess*** in write mode).
- ***lock*** the ***Cache*** with respect to all other modifications, either from the infrastructure or from other application threads. This operation ensures that several operations can be performed on the same ***Cache*** state (i.e., cloning of several objects in a ***CacheAccess***). This operation blocks until the ***Cache*** can be allocated to the calling thread and the waiting time is limited by a time-out (*to_in_milliseconds*). In case the time-out expired before the lock can be granted, an exception (***ExpiredTimeOut***) is raised.
- ***unlock*** the ***Cache***.

Objects attached to the cache are supposed to be garbage-collected when appropriate. There is therefore no specific operation for doing this.

12.That for a sub-access are equivalent.

2.1.6.3.5 CacheListener

CacheListener is an interface that must be implemented by the application in order to be made aware of the arrival of updates on the cache.

CacheListener		
operations		
on_begin_updates		void
on_end_updates		void
on_updates_enabled		void
on_updated_disabled		void

It provides the following methods:

- ***on_begin_updates*** indicates that updates are following. Actual modifications in the cache will be performed only when exiting this method (assuming that ***updates_enabled*** is true).
- ***on_end_updates*** indicates that no more update is foreseen.
- "***on_updates_enabled*** - indicates that the ***Cache*** has switched to automatic update mode. Incoming data will now trigger the corresponding Listeners.
- "***on_updates_disabled*** - indicates that the ***Cache*** has switched to manual update mode. Incoming data will no longer trigger the corresponding Listeners, and will only be taken into account during the next ***refresh*** operation.

In between, the updates are reported on home or selection listeners. Section 2.1.6.4, “Listeners Activation,” on page 2-41 describes which notifications are performed and in what order.

2.1.6.3.6 Contract

Contract is the class that defines which objects will be cloned from the ***Cache*** into the ***CacheAccess*** when the latter is refreshed.

Contract		
attributes		
depth	integer	
scope	ObjectScope	
contracted_object	ObjectRoot	
operations		
set_depth		void
	depth	integer
set_scope		void
	scope	ObjectScope

The public attributes give:

- "The top-level object (***contracted_object***). This is the object that acts as the starting point for the cloning contract.
- "The ***scope*** of the cloning request (i.e., the object itself, or the object with all its (nested) compositions, or the object with all its (nested) compositions and all the objects that are navigable from it up till the specified depth).
- "The ***depth*** of the cloning contract. This defines how many levels of relationships will be covered by the contract (UNLIMITED_RELATED_OBJECTS when all navigable objects must be cloned recursively). The ***depth*** only applies to a RELATED_OBJECT_SCOPE.

It offers methods to:

- "Change the depth of an existing contract (***set_depth***). This change will only be taken into account at the next refresh of the ***CacheAccess***.
- "Change the scope of an existing contract (***set_scope***). This change will only be taken into account at the next refresh of the ***CacheAccess***.

2.1.6.3.7 ObjectHome

For each application-defined class, there is an ***ObjectHome*** instance, which exists to globally represent the related set of instances and to perform actions on it. Actually, ***ObjectHome*** is the root class for generated classes (each one being dedicated to one application-defined class, so that it embeds the related specificity). The name for such a derived class is ***FooHome***, assuming it corresponds to the application-defined class ***Foo***.

A derived ***ObjectHome*** (e.g. a ***FooHome***) has no factory. It is created as an object directly by the natural means in each language binding (e.g., using "new" in C++ or Java).

ObjectHome		
attributes		
class_name	string	
content_filter	string	
registration_index	integer	
auto_deref	boolean	
selections	Selection []	
listener	ObjectListener []	
operations		
get_topic_name		string
	attribute_name	string
get_all_topic_names		string []
set_content_filter		void
	expression	string
set_auto_deref		void
	value	boolean
deref_all		void

underef_all		void
attach_listener		void
	listener	ObjectListener
	concerns_contained_objects	boolean
detach_listener		void
	listener	ObjectListener
create_selection		Selection
	criterion	SelectionCriterion
	auto_refesh	boolean
	concerns_contained_objects	boolean
delete_selection		void
	a_selection	Selection
create_object		ObjectRoot
	access	CacheAccess
create_unregistered_object		ObjectRoot
	access	CacheAccess
register_object		void
	unregistered_object	ObjectRoot
find_object		ObjectRoot
	oid	DLRLOid
	source	CacheBase
get_objects		ObjectRoot[]
	source	CacheBase
get_created_objects		ObjectRoot[]
	source	CacheBase
get_modified_objects		ObjectRoot[]
	source	CacheBase
get_deleted_objects		ObjectRoot[]
	source	CacheBase

The public attributes give:

- the public name of the application-defined class (***class_name***).
- a content filter (***content_filter***) that is used to filter incoming objects. It only concerns subscribing applications; only the incoming objects that pass the content filter will be created in the ***Cache*** and by that ***ObjectHome***. This content filter is given by means of a string and is intended to be mapped on the underlying DCPS infrastructure to provide content-based subscription at DLRL level (see Appendix C for its syntax). The ***content_filter*** attribute is set to NULL by default.
- the ***index*** under which the ***ObjectHome*** has been registered by the ***Cache*** (see ***Cache::register_home*** operation).

- a boolean that indicates whether the state of a DLRL Object should always be loaded into that Object (***auto_deref*** = TRUE) or whether this state will only be loaded after it has been accessed explicitly by the application (***auto_deref*** = FALSE). The ***auto_deref*** attribute is set to TRUE by default.
- the list of attached ***Selection*** (*selections*).
- the list of attached ***ObjectListener*** (*listeners*).

Those last four attributes will be generated properly typed in the derived specific home.

It offers methods to:

- set the ***content_filter*** for that ***ObjectHome*** (***set_content_filter***). As a content filter is intended to be mapped on the underlying infrastructure it can be set only before the ***ObjectHome*** is registered (see ***Cache::register_home***). An attempt to change the filter expression afterwards will raise a ***PreconditionNotMet***. Using an invald filter expression will raise an ***SQLError***.
- set the ***auto_deref*** boolean (***set_auto_deref***).
- ask to load the most recent state of a DLRL Object into that Object for all objects managed by that home (*deref_all*).
- ask to unload all object states from objects that are attached to this home (*underef_all*).
- attach/detach a ***ObjectListener*** (***attach_listener***, ***detach_listener***). When a listener is attached, a boolean parameter specifies, when set to TRUE, that the listener should listen also for the modification of the contained objects (*concerns_contained_objects*).
- create a ***Selection*** (***create_selection***). The ***criterion*** parameter specifies the ***SelectionCriterion*** (either a ***FilterCriterion*** or an ***SelectionCriterion***) to be attached to the ***Selection,*** the ***auto_refresh*** parameter specifies if the ***Selection*** has to be refreshed automatically or only on demand (see ***Selection***) and a boolean parameter specifies, when set to TRUE, that the ***Selection*** is concerned not only by its member objects but also by their contained ones (***concerns_contained_objects***); attached ***SelectionCriterion*** belong to the ***Selection*** that itself belongs to its creating ***ObjectHome***. When creating a ***Selection*** while the DCPS State of the ***Cache*** is still set to INITIAL, a ***PreconditionNotMet*** is raised.
- delete a ***Selection*** (***delete_selection***). This operation deletes the ***Selection*** and its attached ***SelectionCriterion***. If the ***Selection*** was not created by this ***ObjectHome***, a ***PreconditionNotMet*** is raised.
- create a new DLRL object (***create_object***). This operation takes as parameter the ***CacheAccess*** concerned by the creation. The following preconditions must be met: the ***Cache*** must be set to the DCPS State of ENABLED, and the supplied ***CacheAccess*** must writeable. Not satisfying either precondition will raise a ***PreconditionNotMet***.

- pre-create a new DLRL object in order to fill its content before the allocation of the ***oid*** (***create_unregistered_object***); this method takes as parameter the ***CacheAccess*** concerned with this operation. The following preconditions must be met: the ***Cache*** must be set to the DCPS State of ENABLED, and the supplied ***CacheAccess*** must writeable. Not satisfying either precondition will raise a ***PreconditionNotMet***.
- register an object resulting from such a pre-creation (***register_object***). This operation embeds a logic to derive from the object content a suitable ***oid***; only objects created by ***create_unregistered_object*** can be passed as parameter, a ***PreconditionNotMet*** is raised otherwise. If the result of the computation leads to an existing ***oid***, an ***AlreadyExisting*** exception is raised. Once an object has been registered, the fields that make up its identity (i.e. the fields that are mapped onto the keyfields of the corresponding topics) may not be changed anymore.
- retrieve a DLRL object based on its ***oid*** in the in the specified ***CacheBase*** (***find_object***).
- retrieve the name of the topic that contains the value for one attribute (***get_topic_name***). If the DCPS State of the ***Cache*** is still set to INITIAL, a ***PreconditionNotMet*** is raised.
- retrieve the name of all the topics that contain values for all attributes of the class (***get_all_topic_names***). If the DCPS State of the ***Cache*** is still set to INITIAL, a ***PreconditionNotMet*** is raised.
- obtain from a ***CacheBase*** a (typed) list of all objects that match the type of the selected ***ObjectHome*** (***get_objects***). For example the type ***ObjectRoot***[ ] will be substituted by a type ***Foo***[ ] in a ***FooHome***.
- obtain from a ***CacheBase*** a (typed) list of all objects that match the type of the selected ***ObjectHome*** and that have been created, modified or deleted during the last refresh operation (***get_created_objects***, ***get_modified_objects*** and ***get_deleted_objects*** respectively). The type ***ObjectRoot***[ ] will be substituted by a type ***Foo***[ ] in a ***FooHome***.

2.1.6.3.8 ObjectListener

This interface is an abstract root, from which a typed interface will be derived for each application type. This typed interface (named ***FooListener***, if the application class is named ***Foo***), then has to be implemented by the application, so that the application will be made aware of the incoming changes on objects belonging to the ***FooHome***.

ObjectListener		
operations		
on_object_created		boolean
	the_object	ObjectReference
on_object_modified		boolean
	the_object	ObjectRoot
on_object_deleted		boolean
	the_object	ObjectRoot

It is defined with four methods:

- ***on_object_created***, which is called when a new object appears in the ***Cache***; this operation is called with the newly created object (*the_object*).
- ***on_object_deleted***, which is called when an object has been deleted by another participant; this operation is called with the newly deleted object (*the_object*).
- ***on_object_modified***, which is called when the contents of an object changes; this operation is called with the modified object (*the_object*).

Each of these methods must return a boolean. TRUE means that the event has been fully taken into account and therefore does not need to be propagated to other ***ObjectListener*** objects (of parent classes).

See Section 2.1.6.4, “Listeners Activation,” on page 2-41 for a detailed description of how cache, home and selection listeners are called.

2.1.6.3.9 Selection

A ***Selection*** is a mean to designate a subset of the instances of a given ***ObjectHome***, fulfilling a given criterion. This criterion is given by means of the attached ***SelectionCriterion***.

Selection		
attributes		
criterion	SelectionCriterion	
auto_refresh	boolean	
concerns_contained	boolean	
members	ObjectRoot[]	
listener	SelectionListener	
operations		
set_listener		SelectionListener
	listener	SelectionListener
refresh		void

Actually, the ***Selection*** class is a root from which are derived classes dedicated to application classes (for an application class named ***Foo***, ***FooSelection*** will be derived).

It has the following attributes:

- the corresponding ***SelectionCriterion*** (***criterion***). It is given at ***Selection*** creation time (see ***ObjectHome::create_selection***).
- a boolean ***auto_refresh*** that indicates if the ***Selection*** has to be refreshed at each incoming modification (TRUE) or only on demand (FALSE). It is given at ***Selection*** creation time (see ***ObjectHome::create_selection***).

- a boolean *concerns_contained* that indicates whether the ***Selection*** considers he modification of one of its members based on its content only (FALSE) or based on its content or the content of its contained objects (TRUE). It is given at ***Selection*** creation time (see ***ObjectHome::create_selection***).
- the list of the objects that are part of the selection (***members***).
- attached ***listener***.

It offers the methods to:

- set the ***SelectionListener*** (***set_listener***), that will be triggered when the composition of the selection changes, as well as if the members are modified. set_listener returns the previously set listener if any; ***set_listener*** called with a NULL parameter discards the current listener.
- request that the ***Selection*** updates its members (***refresh***).

The ***SelectionListener*** is activated when the composition of the ***Selection*** is modified as well as when one of its members is modified. A member can be considered as modified, either when the member is modified or when that member or one of its contained objects is modified (depending on the value of *concerns_contained*). Modifications in the ***Selection*** are considered with respect to the state of the ***Selection*** last time is was examined, for instance:

- at each incoming updates processing, if *auto_refresh* is TRUE.
- at each explicit call to ***refresh***, if *auto_refresh* is FALSE.

2.1.6.3.10 SelectionCriterion

An ***SelectionCriterion*** is an object (attached to a ***Selection***) that gives the criterion to be applied to make the ***Selection***.It is the abstract base-class for both the ***FilterCriterion*** and the ***QueryCriterion***.

SelectionCriterion	
attributes	
`kind`	`SelectionCriteria`
no operations	

It has one attribute (***kind***) that describes whether a SelectionCriterion instance represents a ***FilterCriterion*** or a ***QueryCriterion***.

2.1.6.3.11 FilterCriterion

FilterCriterion is a specialization of ***SelectionCriterion*** that performs the object check based on a user-defined filter algorithm.

FilterCriterion : SelectionCriterion		
no attributes		
operations		
check_object		boolean
	an_object	ObjectRoot
	membership_state	enum MembershipState

It offers a method to:

- check if an object passes the filter – return value is TRUE – or not – return value is FALSE (***check_object***). This method is called with the first parameter set to the object to be checked and the second parameter set to indicate whether the object previously passed the filter (membership_state). The second parameter (which is actually an enumeration with three possible values - UNDEFINED_MEMBERSHIP, ALREADY_MEMBER and NOT_MEMBER) is useful when the ***FilterCriterion*** is attached to a ***Selection*** to allow writing optimized filters.

The ***FilterCriterion*** class is a root from which are derived classes dedicated to application classes (for an application class named ***Foo***, ***FooFilter*** will be derived).

FooFilter is itself a base class that may be derived by the application in order to provide its own ***check_object*** algorithm. The default provided behavior is that ***check_object*** always return TRUE.

2.1.6.3.12 QueryCriterion

QueryCriterion is a specialization of ***SelectionCriterion*** that performs the object check based on a query expression.

QueryCriterion : SelectionCriterion		
attributes		
expression	string	
parameters	string []	
operations		
set_query		boolean
	expression	string
	arguments	string []
set_parameters		boolean
	arguments	string []

The query is made of an ***expression*** and of ***parameters*** that may parameterize the ***expression*** (the number of ***parameters*** must fit with the values required by the ***expression***). See Appendix C for the syntax of an ***expression*** and its ***parameters***.

It offers methods to:

- set the value of the ***expression*** and its ***parameters*** (***set_query)***; a TRUE return value indicates that they have been successfully changed.
- set the values of the parameters (***set_parameters***). The number of ***parameters*** must fit with the values required by the ***expression***. A TRUE return value indicates that they have been successfully changed.

After a successful call to one of those methods the owning ***Selection*** is refreshed if its ***auto_refresh*** is TRUE.

2.1.6.3.13 SelectionListener

This interface is an abstract root, from which a typed interface will be derived for each application type. This typed interface (named ***FooSelectionListener***, if the application class is named ***Foo***) has to be implemented by the application in order to be made aware of the incoming changes on objects belonging to a ***FooSelection***.

SelectionListener		
operations		
on_object_in		void
	the_object	ObjectRoot
on_object_out		void
	the_object	ObjectRoot
on_object_modified		void
	the_object	ObjectRoot

It is defined with three methods:

- ***on_object_in***, which is called when an object enters the ***Selection***.
- ***on_object_out***, which is called when an object exits the ***Selection***.
- ***on_object_modified***, which is called when the contents of an object belonging to the ***Selection*** changes.

Section 2.1.6.4, “Listeners Activation,” on page 2-41 includes a detailed description of how cache, home, and selection listeners are called.

2.1.6.3.14 ObjectRoot

ObjectRoot is the abstract root for any DLRL class. It brings all the properties that are needed for DLRL management. ***ObjectRoot*** are used to represent either objects that are in the ***Cache*** (also called primary objects) or clones that are attached to a ***CacheAccess*** (also called secondary objects). Secondary objects refer to a primary one with which they share the ***ObjectReference***.

.

ObjectRoot		
attributes		
oid	DLRLOid	
read_state	ObjectState	
write_state	ObjectState	
object_home	ObjectHome	
owner	CacheBase	
operations		
destroy		void
is_modified		boolean
	scope	ObjectScope
which_contained_modified		RelationDescription[]

Its public attributes[13] give:

- the identity of the object (***oid***);
- its lifecycle states (***read_state*** and ***write_state***);
- its related home (***object_home***);
- the cache it belongs to (***owner***), this can be either a ***Cache*** or a ***CacheAccess***.

It offers methods to:

- mark the object for destruction (***destroy***), to be executed during a write operation. If the object is not located in a writeable ***CacheAccess***, a ***PreconditionNotMet*** is raised.
- see if the object has been modified by incoming modifications (***is_modified***). ***is_modified*** takes as parameter the scope of the request (i.e., only the object contents, the object and its component objects, the object and all its related objects). In case the object is newly created, this operation returns FALSE; 'incoming modifications' should be understood differently for a primary object and for a clone object.
  - For a primary object, they refer to incoming updates (i.e., coming from the infrastructure).
  - For a secondary object (cloned), they refer to the modifications applied to the object by the last CacheAccess::refresh operation.

13. It is likely that other attributes are needed to manage the objects (i.e., a content version, a reference count...); however these are implementation details not part of the specification.

- get which contained objects have been modified (***which_contained_modified***). This method returns a list of descriptions for the relations that point to the modified objects (each description includes the name of the relation and if appropriate the index or key that corresponds to the modified contained object).

In addition, application classes (i.e., inheriting from ObjectRoot), will be generated with a set of methods dedicated to each shared attribute (including single- and multi-relation attributes):

- ***get_<attribute>***, read accessor to the attribute - this accessor will embed whatever is needed to properly get the data.
- ***set_<attribute>***, write accessor for the attribute - this accessor will embed whatever is needed to further properly ***write*** the data to the publishing infrastructure (in particular, it will take note of the modification). Since the identity of DLRL Objects that are generated using predefined mapping (i.e. with a keyDescription content of "NoOid") is determined by the value of its key fields, changing these key fields means changing their identity. For this reason these keyfields are considered read-only: any attempt to change them will raise a ***PreconditionNotMet***. The only exception to this rule is when locally created objects have not yet been registered and therefore do not have an identity yet.
- ***is_<attribute>_modified***, to get if this attribute has been modified by means of incoming modifications (cf. method ***is_modified***).

A ***Cache*** Object represents the global system state. It has a ***read_state*** whose transitions represent the updates as they are received by the DCPS. Since ***Cache*** Objects cannot be modified locally, they have no corresponding ***write_state*** (i.e. their ***write_state*** is set to VOID). State transitions occur between the start of an update round and the end of of an update round. When in automatic updates mode, the start of the update round is signaled by the invocation of the ***on_begin_updates*** callback of the ***CacheListener***, while the end of an update round is signaled by the invocation of the ***on_end_updates*** callback of the ***CacheListener***. When in manual update mode, the start of an update round is defined as the start of a refresh operation, while the end of an update round is defined as the invocation of the next refresh operation.

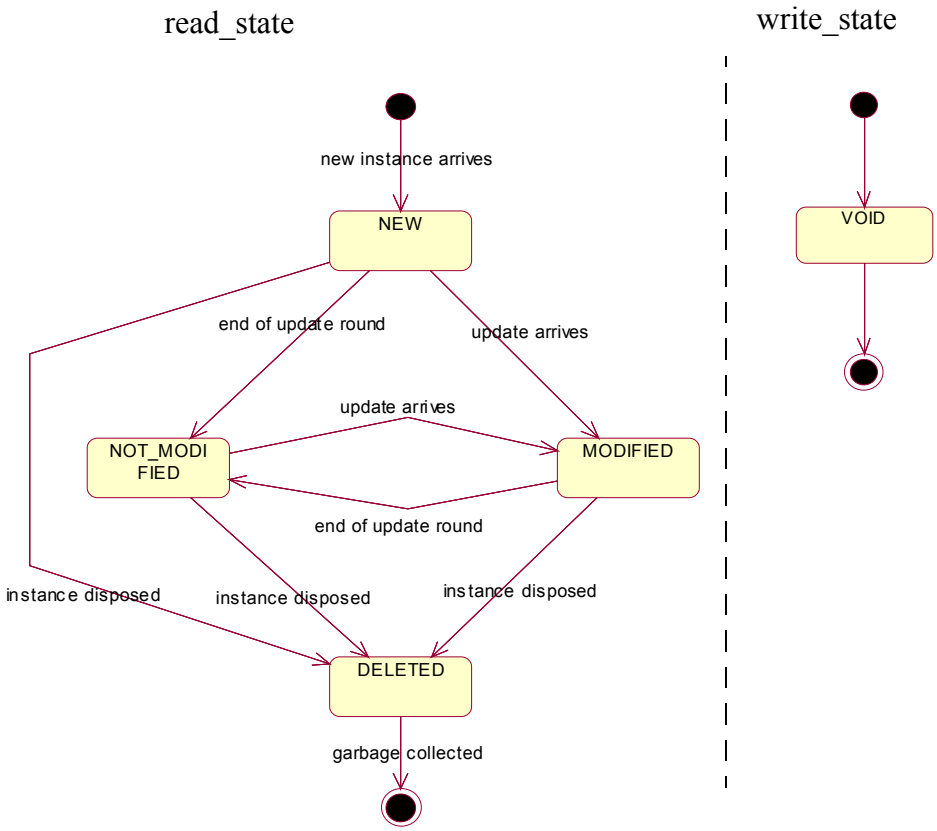
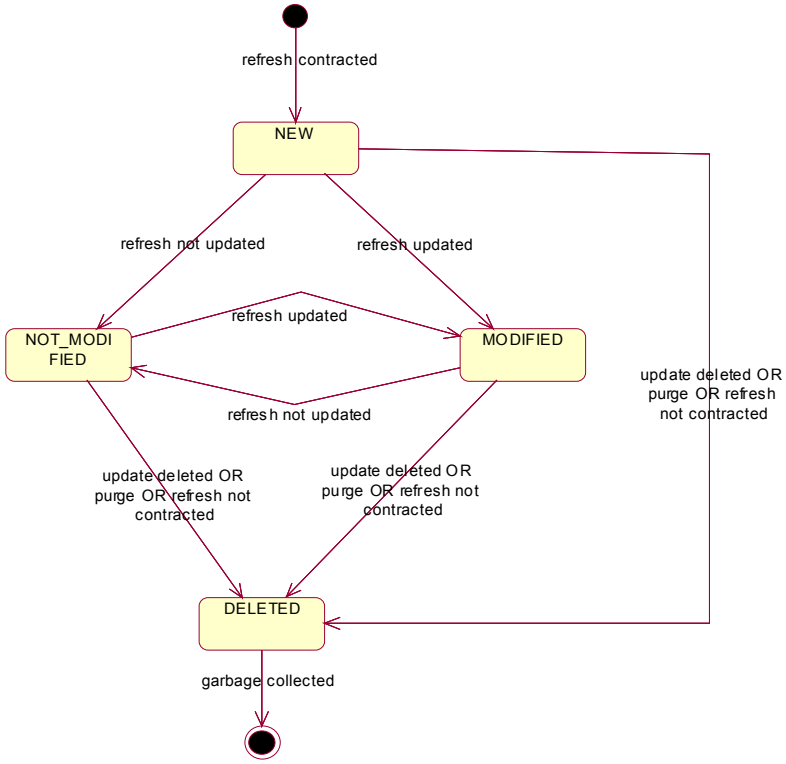


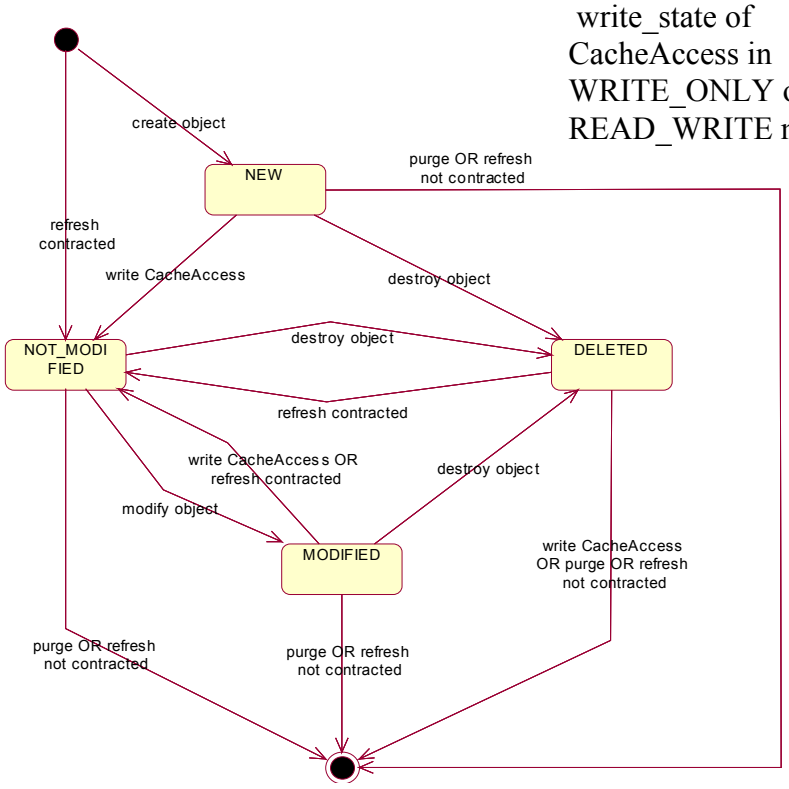
Figure 2-5 read_state and write_state of a Cache object

A ***CacheAccess*** Object represents either represents a temporary system state (a so-called 'snapshot' of the ***Cache***) when in READ_ONLY mode, or it represents an intended system state when in WRITE_ONLY or READ_WRITE mode. In READ_ONLY mode, a ***CacheAccess*** object has no ***write_state*** (it is VOID, not depicted), while in WRITE_ONLY mode it has no ***read_state*** (it is VOID, not depicted). Transitions of the ***read_state*** occur during an update round (caused by invocation of the refresh method) , or when the CacheAccess is purged. Changes of the ***write_state*** are caused by either local modifications (can be done on any time), by commiting the local changes to the system (during a write operation), by purging the ***CacheAccess*** or by starting a new update round (by invoking the ***refresh*** method and thus rolling back any uncommitted changes). Since a ***refresh*** operation validates contracts, and both these contracts and the relationships between their targeted objects may change, two results are possible: an object can be contracted as a result of the ***refresh*** operation, thus (re-)appearing in the ***CacheAccess***, or an object can not be contracted as a result of a ***refresh*** operation, thus disappearing from a ***CacheAccess.***

Figure 2-6 ead_state and write_state of a CacheAccess object



write_state of CacheAccess in WRITE_ONLY or READ_WRITE mode



2.1.6.3.15 Collection

This class is the abstract root for all collections (lists and maps).

Collection	
no attributes	
length	integer
values	undefined [] (e.g. of type ObjectRoot or Primitive type)

It provides the following attributes:

- ***length*** - the length of the collection.
- ***values*** - a list of all values contained in the ***Collection.***

2.1.6.3.16 List

This class is the abstract root for all the lists. Concrete list classes will be derived, in order to provide typed lists (those classes will be named ***FooList***, assuming that ***Foo*** is the type of one item).

List : Collection		
no attributes		
operations		
remove		void
added_elements		integer[]
removed_elements		integer[]
modified_elements		integer[]
add		void
	value	undefined (e.g. of type ObjectRoot or Primitive type)
put		void
	index	integer
	value	undefined (e.g. of type ObjectRoot or Primitive type)
get		undefined (e.g. of type ObjectRoot or Primitive type)
	index	integer

It provides the following methods:

- ***"remove*** - to remove the item with the highest index from the collection.
- ***"added_elements*** **-** to get a list that contains the indexes of the added elements.

- ***"removed_elements*** - to get a list that contains the indexes of the removed elements.
- ***"modified_elements -*** to get a list that contains the indexes of the modified elements.
- ***"add -*** to add an item to the the end of the list.
- ***"put -*** to put an item in the collection at a specified index.
- ***"get -*** to retrieve an item in the collection (based on its index).

2.1.6.3.17 Set

This class is the abstract root for all setss. Concrete ***Set*** classes will be derived, in order to provide typed sets (those classes will be named ***FooSet***, assuming that ***Foo*** is the type of one item).

Set : Collection		
no attributes		
operations		
added_elements		undefined (e.g. of type ObjectRoot or Primitive type)
removed_elements		undefined (e.g. of type ObjectRoot or Primitive type)
contains		boolean
	value	undefined (e.g. of type ObjectRoot or Primitive type)
add		void
	value	undefined (e.g. of type ObjectRoot or Primitive type)
remove		void
	value	undefined (e.g. of type ObjectRoot or Primitive type)

- It provides the following methods:
- "***add*** - to add an element to the ***Set***. If the specified element was already contained in the ***Set***, the operation is ignored.
- "***remove*** - to remove an element from the Set. If the specified element is not contained in the ***Set***, the operation is ignored.
- "***contains*** - returns whether the specified value is already contained in the ***Set*** (true) or not (false).
- "***added_elements*** - to return the elements added in the last update round.

- "***removed_elements*** - to return the elements removed in the last update round

2.1.6.3.18 StrMap

This class is the abstract root for all the maps with string keys. Concrete map classes will be derived, in order to provide typed maps (those classes will be named ***FooStrMap***, assuming that ***Foo*** is the type of one item).

StrMap : Collection		
attributes		
keys	string[]	
operations		
remove		void
	key	string
added_elements		string[]
removed_elements		string[]
modified_elements		string[]
put		void
	key	string
	value	undefined (e.g. of type ObjectRoot or Primitive type)
get		undefined (e.g. of type ObjectRoot or Primitive type)
	key	string

The public attributes give:

- "***keys*** - a list that contains all the keys of the items belonging to the map.

It provides the following methods:

- "***remove*** - to remove an item from the collection.
- "***added_elements*** - to get a list that contains the keys of the added elements.
- "***removed_elements*** - to get a list that contains the keys of the removed elements.
- "***modified_elements*** -to get a list that contains the keys of the modified elements.
- "***put*** - to put an item in the collection.
- "***get*** - to retrieve an item in the collection (based on its key).

2.1.6.3.19 IntMap

This class is the abstract root for all the maps with integer keys. Concrete map classes will be derived, in order to provide typed maps (those classes will be named ***FooIntMap***, assuming that ***Foo*** is the type of one item).

IntMap : Collection		
attributes		
keys	string[]	
operations		
remove		void
	key	integer
added_elements		integer[]
removed_elements		integer[]
modified_elements		integer[]
put		void
	key	integer
	value	undefined (e.g. of type ObjectRoot or Primitive type)
get		undefined (e.g. of type ObjectRoot or Primitive type)
	key	integer

The public attributes give:

- "keys - a list that contains all the keys of the items belonging to the map.

It provides the following methods:

- "***remove*** - to remove an item from the collection.
- "***added_elements*** - to get a list that contains the keys of the added elements.
- "***removed_elements*** - to get a list that contains the keys of the removed elements.
- "***modified_elements*** - to get a list that contains the keys of the modified elements.
- "***put*** - to put an item in the collection.
- "***get*** - to retrieve an item in the collection (based on its key).

2.1.6.4 Listeners Activation

As described in Section 2.1.6.2, “DLRL Entities,” on page 2-15, there are three kinds of listeners that the application developer may implement and attach to DLRL entities: ***CacheListener***, ***ObjectListener***, and ***SelectionListener***. All these listeners are a means for the application to attach specific application code to the arrival of some events. They are therefore only concerned with incoming information.

This section presents how these listeners are triggered (i.e., which ones, on which events, and in which order).

2.1.6.4.1 General Scenario

Incoming updates[14] are usually a set of coherent individual updates that may be object creations, object deletions, and object modifications.

This set of updates is managed as follows:

- First, all the ***CacheListener::start_updates*** operations are triggered; the order in which these listeners are triggered is not specified.
- Then all the updates are actually applied in the cache[15]. When an object is modified, several operations allow to get more precisely which parts of the object are concerned (see ***ObjectRoot::is_modified*** operations as well as the operations for ***Collection***, namely, ***is_modified***, ***how_many_added***, ***how_many_removed***, ***removed_values***, and ***which_added***); these operations can be called in the listeners.
- Then, the suitable object and selection listeners are triggered, depending on each individual update (see the following sections).
- Finally all the ***CacheListener::end_updates*** operations are triggered and the modification states of the updated objects is cleaned; the order in which these listeners are triggered is not specified.

2.1.6.4.2 Object Creation

When an individual update reports an object creation, the following listeners are activated:

- First, the ***ObjectListener*** listeners suitable to that object are searched and their ***on_object_created*** operations triggered. The search follows the inheritance structure starting with the more specific ***ObjectHome*** (e.g., ***FooHome***, if the object is typed ***Foo***) to ***ObjectRoot***. The search is stopped when all ***on_object_created*** operations return true at one level; inside one level, the triggering order is not specified.
- Then, all the ***Selection*** objects that are concerned with that kind of object (e.g., the ***FooSelection*** and above in the inheritance hierarchy) are checked to see if that new object is becoming a member of the selection. In case it is true, the attached ***SelectionListener::on_object_in*** is triggered.

2.1.6.4.3 Object Modification

When an individual update reports an object modification, the following listeners are activated:

---

14. Whether those incoming updates are transmitted to the DLRL layer by means of DCPS listeners or by means of wait sets and conditions is not discussed here: this is an implementation detail.

15. If an object is deleted, its state is set as DELETED; it will be actually removed when there are no more references to it.

- First, all the ***Selection*** objects that are concerned with that kind of object (e.g., the ***FooSelection*** and above in the inheritance hierarchy, assuming that the object is of type ***Foo***) are checked to see if that new object is:
  - becoming a member of the selection. If so, the attached ***SelectionListener::on_object_in*** is triggered.
  - already and still part of the selection. If so, the attached ***SelectionListener::on_object_modified*** is triggered.
  - leaving the selection. If so, the attached ***SelectionListener::on_object_out*** is triggered.
- Then, the ***ObjectListener*** listeners suitable to that object are searched and their ***on_object_modified*** operations triggered. The search follows the inheritance structure starting with the more specific ***ObjectHome*** (e.g., ***FooHome***, if the object is typed ***Foo***) to ***ObjectRoot***. The search is stopped when all ***on_object_modified*** operations return true at one level; inside one level, the triggering order is not specified.

2.1.6.4.4 Object Deletion

When an individual update reports an object deletion, the following listeners are activated.

- First, all the ***Selection*** objects that are concerned with that kind of object (e.g., the ***FooSelection*** and above in the inheritance hierarchy, assuming that the object is of type ***Foo***) are checked to see if that new object was part of the selection. If so, the attached ***SelectionListener::on_object_out*** is triggered.
- Then, the ***ObjectListener*** listeners suitable to that object are searched and their ***on_object_deleted*** operations triggered. The search follows the inheritance structure starting with the more specific ***ObjectHome*** (e.g., ***FooHome***, if the object is typed ***Foo***) to ***ObjectRoot***. The search is stopped when all ***on_object_deleted*** operations return true at one level; inside one level, the triggering order is not specified.

2.1.6.5 Cache Accesses Management

Cache accesses are a means to perform read or write operations in isolation from other object modifications. The two following subsections present typical use scenarios.

It should be noted that, even though a sensible design is to create a ***CacheAccess*** per thread, DLRL does not enforce this rule by any means.

2.1.6.5.1 Read Mode

The typical scenario for read mode is as follows:

1. Create the ***CacheAccess*** for read purpose (***Cache::create_access***).
2. Attach some cloning contracts to it (CacheAccess::create_contract).
3. Execute these contracts (CacheAccess::refresh).

4. Consult the clone objects and navigate amongst them (plain access to the objects). These objects are not subject to any incoming notifications.
5. Purge the cache (***CacheAccess::purge***); step 2 can be started again.
6. Eventually, delete the ***CacheAccess*** (***Cache::delete_access***).

2.1.6.5.2 Write Mode

The typical scenario for write mode is as follows:

1. Create the ***CacheAccess*** for write purpose (***Cache::create_access***).
2. Clone some objects in it (***ObjectRoot::clone*** or ***clone_object***).
3. Refresh them (***CacheAccess::refresh***).
4. If needed create new ones for that ***CacheAccess*** (***ObjectHome:: create_object***).
5. Modify the attached (plain access to the objects).
6. Write the modifications into the underlying infrastructure (***CacheAccess::write***).
7. Purge the cache (***CacheAccess::purge***); step 2 can be started again.
8. Eventually, delete the ***CacheAccess*** (***Cache::delete_access***).

2.1.6.6 Generated Classes

Assuming that there is an application class named ***Foo*** (that will extend ***ObjectRoot***), the following classes will be generated:

- FooHome : ObjectHome
- FooListener : ObjectListener
- FooSelection : Selection
- FooSelectionListener : SelectionListener
- FooFilter : FilterCriterion
- FooQuery : FooFilter, QueryCriterion
- And for relations to ***Foo*** objects (assuming that these relations are described in the applicative mode – note also that the actual name of these classes will be indicated by the application):
  - “FooRelation” : RefRelation
  - “FooListRelation” : ListRelation
  - “FooStrMapRelation” : StrMapRelation
  - “FooIntMapRelation” : IntMapRelation

2.2 OMG IDL Platform Specific Model (PSM)

This section provides a mapping suitable for CORBA platforms. It is described by means of IDL constructs that can be used by an application in order to interact with the services; this is described in Section 2.2.1, “Run-time Entities,” on page 2-45.

This section also specifies the generation process (specializing the abstract one presented on Figure 2-3 : DLRL Generation Process); in particular, the following are described:

- How the application introduces its application classes (“Model Description” in Figure 2-3).
- How the application adds indication to properly generate the DLRL entities as well as the resulting enhanced application constructs (“Model Tags” in Figure 2-3).

This process is described in Section 2.2.2, “Generation Process,” on page 2-62.

2.2.1 Run-time Entities

2.2.1.1 Mapping Rules

Rationale to define DLRL entities mapping is slightly different from what ruled the DCPS mapping, mainly because this layer does not target C language. Therefore, valuetypes or exceptions have been considered as suitable at the DLRL level, while they have been rejected for DCPS.

In summary, there are two kinds of DLRL entities:

1. Entities that are access points to servicing objects (e.g., ***Cache***).
2. Entities that are application objects (i.e., whose aim is to be distributed), or parts of them.

Entities belonging to the first category are modeled as IDL local interfaces. Entities belonging to the second one are modeled as IDL valuetypes.

The choice for valuetypes has been driven by two main reasons:

- It is the IDL construct that fits best with the concept of DLRL objects.
- It offers a means to differentiate private from public attributes.

Error reporting has been modeled by use of exceptions, with the following rule:

- When a real error that will affect the future behavior is reported (e.g., passing of a wrong parameter), an exception is raised.
- When this ‘error’ is actually a warning in the sense that behavior will not be affected (e.g., an attempt to remove something from a list where it is not, or no more), a return value is used instead.

The language implementation of the ***CacheFactory*** interface should have the static operation ***get_instance*** described in Section 2.1.6.3.1, “CacheFactory,” on page 2-19. This operation does not appear in the IDL ***CacheFactory*** interface, as static operations cannot be expressed in IDL

The IDL PSM introduces a number of types that are intended to be defined in a native way. As these are opaque types, the actual definition of the type does not affect portability and is implementation dependent. For completeness the names of the types appear as typedefs in the IDL and a #define with the suffix "_TYPE_NATIVE" is used as a place-holder for the actual type. The type used in the IDL by this means is not normative and an implementation is allowed to use any other type, including non-scalar (i.e., structured types).

Exceptions in DLRL will be mapped according to the default language mapping rules, except for the ***AlreadyDeleted*** exception. Since this exception can be raised on all methods and attributes (which is not possible to specify in IDL versions older than 3.0), it is not explicitly mentioned in the raise clause of each operation. Implementors may choose to map it onto an exception type that does not need to be caught explicitly, simplifying the DLRL code significantly.

2.2.1.2 IDL Description

This IDL is split in two sections:

- IDL for the generic DLRL entities
- Implied IDL

2.2.1.2.1 Generic DLRL Entities

```
#include "dds_dcps.idl"

#define DLRL_OID_TYPE_NATIVE long

module DDS {


// Type definitions
// ================

// Scope of action
// ---------------

enum ReferenceScope {
SIMPLE_CONTENT_SCOPE,          // only the reference content
REFERENCED_CONTENTS_SCOPE      // + referenced contents
};

enum ObjectScope {
SIMPLE_OBJECT_SCOPE,           // only the object
```

```
CONTAINED_OBJECTS_SCOPE,       // + contained objects
RELATED_OBJECTS_SCOPE          // + all related objects
};

// State of the underlying infrastructure
// --------------------------------------

enum DCPSState {
   INITIAL,
   REGISTERED,
   ENABLED
   };

// Usage of the Cache
// ------------------

enum CacheUsage {
   READ_ONLY,
   WRITE_ONLY,
   READ_WRITE
   };

// Object State
// ------------
enum ObjectState {
   OBJECT_VOID,
   OBJECT_NEW,
   OBJECT_NOT_MODIFIED,
   OBJECT_MODIFIED,
   OBJECT_DELETED
};

// OID
// ---

struct DLRLOid {
   DLRL_OID_TYPE_NATIVE value[3];
   };


// Miscellaneous
// ------------

typedef sequence<long>     LongSeq;

typedef string         ClassName;
typedef string         CacheName;
typedef string         RelationName;


// Exceptions
```

```
// ==========

exception DCPSError { string message; };
exception BadHomeDefinition { string message; };
exception NotFound { string message; };
exception AlreadyExisting { string message; };
exception AlreadyDeleted { string message; };
exception PreconditionNotMet { string message; };
exception NoSuchElement { string message; };
exception SQLError { string message; };

// DLRL Entities
// =============

/*******************
 * Forward References
 *******************/

valuetype ObjectRoot;
typedef sequence<ObjectRoot> ObjectRootSeq;

local interface ObjectHome;
typedef sequence<ObjectHome> ObjectHomeSeq;

local interface ObjectListener;
typedef sequence<ObjectListener> ObjectListenerSeq;

local interface Selection;
typedef sequence<Selection> SelectionSeq;

local interface CacheBase;
typedef sequence<CacheBase> CacheBaseSeq;

local interface CacheAccess;
typedef sequence<CacheAccess> CacheAccessSeq;

local interface CacheListener;
typedef sequence<CacheListener> CacheListenerSeq;

local interface Cache;

local interface Contract;
typedef sequence<Contract> ContractSeq;

/****************************************************
 * ObjectListener : Root for Listeners to be attached to
 *      Home objects
 ****************************************************/

local interface ObjectListener {
   boolean on_object_created (
```

```
      in ObjectRoot the_object);

   /**** will be generated with the proper Foo type* in the derived
     *     FooListener
     *  boolean on_object_modified (
     *          in ObjectRoot the_object);
     ****/

   boolean on_object_deleted (
      in ObjectRoot the_object);
   };


/*********************************************************
 * SelectionListener : Root for Listeners to be attached to
 *     Selection objects
 *********************************************************/

local interface SelectionListener {
   /***
    * will be generated with the proper Foo type
    * in the derived FooSelectionListener
    *
   void on_object_in (
      in ObjectRoot the_object);
   void on_object_modified (
      in ObjectRoot the_object);
    *
    ***/
   void on_object_out (
      in ObjectRoot the_object);
   };


/*********************************************************
 * CacheListener : Listener to be associated with a Cache
 *********************************************************/

local interface CacheListener {
   void on_begin_updates ();
   void on_end_updates ();
   void on_updates_enabled( );
   void on_updates_disabled( );
};


/*********************************************************
 * Contract : Control objects cloned on a CacheAccess refresh
 *********************************************************/

local interface Contract {
```

```
    readonly attribute long depth;
    readonly attribute ObjectScope scope;
    readonly attribute ObjectRoot contracted_object.

    void set_depth(
in long depth);
    void set_scope(
        in ObjectScope scope);
};


/*****************************************
 * ObjectRoot : Root fot the shared objects
 *****************************************/
enum RelationKind {
    REF_RELATION,
    LIST_RELATION,
    INT_MAP_RELATION,
    STR_MAP_RELATION};

valuetype RelationDescription {
    public RelationKind    kind;
    public RelationName    name;
    };
valuetype ListRelationDescription : RelationDescription {
    public long index;
    };
valuetype IntMapRelationDescription : RelationDescription {
    public long key;
    };
valuetype StrMapRelationDescription : RelationDescription {
    public string key;
    };
typedef sequence<RelationDescription> RelationDescriptionSeq;

typedef short   RelatedObjectDepth;
const RelatedObjectDepth UNLIMITED_RELATED_OBJECTS  = -1;

valuetype ObjectRoot {

    // State
    // -----
    private DLRLOid     m_oid;
    private ClassName   m_class_name;

    // Attributes
    // ----------
    readonly attribute  DLRLOid          oid;
    readonly attribute  ObjectState      read_state;
    readonly attribute  ObjectState      write_state;
    readonly attribute  ObjectHome       object_home;
```

```
    readonly attribute  ClassName       class_name;
    readonly attribute  CacheBase       owner;

    // Operations
    // ----------
    void destroy ()
        raises (
            PreconditionNotMet);
    boolean is_modified (
        in ObjectScope scope);
    RelationDescriptionSeq which_contained_modified ();
    };

/***********************************************
* SelectionCriterion: Root of all filters and queries
***********************************************/
enum CriterionKind {
    QUERY,
    FILTER
};

local interface SelectionCriterion {
    readonly attribute CriterionKind kind;
};

/***********************************************
* FilterCriterion: Root of all the objects filters
***********************************************/
enum MembershipState {
    UNDEFINED_MEMBERSHIP,
    ALREADY_MEMBER,
    NOT_MEMBER
};

local interface FilterCriterion : SelectionCriterion {
    /***
    * Following method will be generated properly typed
    * in the generated derived classes
    *
    boolean check_object (
    in ObjectRoot an_object,
    in MembershipState membership_state);
    *
    ***/
};
/*****************************************************
* QueryCriterion : Specialized SelectionCriterion to make a
* Query
*****************************************************/
local interface QueryCriterion : SelectionCriterion {
    // Attributes
```

```
    // ---------
    readonly attribute string expression;
    readonly attribute StringSeq parameters;
    //--- Methods
    boolean set_query (
        in string expression,
        in StringSeq parameters) raises (SQLError);
    boolean set_parameters ( in StringSeq parameters ) raises (SQLError);
};


/************************************************************
 * Selection : Root of all the selections (dynamic subsets)
 ************************************************************/

local interface Selection {

    // Attributes
    // ----------
    readonly attribute boolean          auto_refresh;
    readonly attribute boolean          concerns_contained;

    /***
     * Following attributes will be generated properly typed
     * in the generated derived classes
     *
    readonly attribute SelectionCriterion   criterion;
    readonly attribute ObjectRootSeq        members;
    readonly attribute SelectionListener    listener;
     *
     */

    // Operations
    // ----------
    /***
     * Following method will be generated properly typed
     * in the generated derived classes
     *
    SelectionListener set_listener (
        in SelectionListener listener);
     *
     ***/
    void refresh ();
    };


/*********************************************************************
 * ObjectHome : Root of all the representatives of applicative classes
 *********************************************************************/

local interface ObjectHome {
```

```
// Attributes
// ----------
readonly attribute string          name;   // Shared name of the class
readonly attribute string          content_filter;
readonly attribute ObjectHome      parent;
readonly attribute ObjectHomeSeq   children;
readonly attribute unsigned long   registration_index;
readonly attribute boolean         auto_deref;

/***
 * Following attributes will be generated properly typed
 * in the generated derived classes
 *
readonly attribute SelectionSeq       selections;
readonly attribute ObjectListenerSeq  listeners;
 *
 ***/

// Operations
// ----------

void set_content_filter (
   in string expression)
   raises (
      SQLError,
      PreconditionNotMet);

void set_auto_deref (
   in boolean value);
void deref_all();
void underef_all ();

//--- Relations to topics

string get_topic_name (
   in string attribute_name)
   raises (
      PreconditionNotMet);
StringSeq get_all_topic_names ()
   raises (
      PreconditionNotMet);


// --- Listener management

/***
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
  void attach_listener (
```

```
        in ObjectListener listener,
        in boolean concerns_contained_objects);
  void detach_listener (
        in ObjectListener listener);
 *
 ***/

// --- Selection management

/***
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
Selection create_selection(
   in SelectionCriterion criterion,
   in boolean auto_refresh,
   in boolean concerns_contained_objects )
   raises (
      PreconditionNotMet );
void delete_selection (
   in Selection a_selection)
   raises (
      PreconditionNotMet);
 *
 ***/

// --- Object management

/***
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
ObjectRoot create_object(
   in CacheAccess access)
   raises (
      PreconditionNotMet);
ObjectRoot create_unregistered_object (
   in CacheAccess access)
   raises (
      PreconditionNotMet);
void register_object (
   in ObjectRoot   unregistered_object)
   raises (
      AlreadyExisting,
      PreconditionNotMet);

ObjectRoot find_object (
   in DLRLOid  oid,
   in CacheBase source)
   raises (
      NotFound);
```

```
    ObjectRootSeq get_objects (
       in CacheBase source);
    ObjectRootSeq get_created_objects (
       in CacheBase source);
    ObjectRootSeq get_modified_objects (
       in CacheBase source);
    ObjectRootSeq get_deleted_objects (
       in CacheBase source);


     *
     ***/
    };

/***********************
 * Collection operations
 ***********************/
abstract valuetype Collection {

    readonly attribute long length;

    /***
     * The following methods will be generated properly typed
     * in the generated derived classes
     *
     readonly attribute ObjectRootSeq values;
      *
      ***/
};

abstract valuetype List : Collection {

    void remove( );
    LongSeq added_elements( );
    LongSeq removed_elements( );
    LongSeq modified_elements( );

    /***
     * The following methods will be generated properly typed
     * in the generated derived classes
     *
     void add( in ObjectRoot value );
     void put( in long key, in ObjectRoot value );
     ObjectRoot get( in long key );
      *
      ***/
};

valuetype Set : Collection {
    /***
```

```
     * The following methods will be generated properly typed in
     * the generated derived classes.
     *
    ObjectRootSeq added_elements( );
    ObjectRootSeq removed_elements( );
    boolean contains( ObjectRoot value );
    void add( ObjectRoot value );
    void remove( ObjectRoot value );
     *
     ***/
};

abstract valuetype StrMap : Collection {

    readonly attribute StringSeq keys;
    void remove( in string key );
    StringSeq added_elements( );
    StringSeq removed_elements( );
    StringSeq modified_elements( );

    /***
     * The following methods will be generated properly typed
     * in the generated derived classes
     *
     void put( in string key, in ObjectRoot value );
     ObjectRoot get( in string key );
      *
      ***/
};

abstract valuetype IntMap : Collection {

    readonly attribute LongSeq keys;
    void remove( in long key );
    LongSeq added_elements( );
    LongSeq removed_elements( );
    LongSeq modified_elements( );

    /***
     * The following methods will be generated properly typed
     * in the generated derived classes
     *
     void put( in long key, in ObjectRoot value );
     ObjectRoot get( in long key );
      *
      ***/
};

/************************************************************
 * CacheBase : Base class to CacheAccess and Cache
 ************************************************************/
```

```
enum CacheKind {
   CACHE_KIND,
   CACHEACCESS_KIND
};

local interface CacheBase {
   readonly attribute CacheUsage cache_usage;
   readonly attribute ObjectRootSeq objects;
   readonly attribute CacheKind kind;

   void refresh( )  raises (DCPSError);
};

/************************************************************
 * CacheAccess : Manager of the access of a subset of objects
 *     (cloned) from a Cache
 ************************************************************/

local interface CacheAccess : CacheBase {

   // Attributes
   // ==========
   readonly attribute Cache             owner;
   readonly attribute ContractSeq       contracts;
   readonly attribute StringSeq         type_names;

   // Operations
   // ==========
   void write ()
      raises (
         ReadOnlyMode,
         DCPSError);
   void purge ();
   void create_contract(
      in ObjectRoot object,
      in ObjectScope scope, in long depth )
      raises (PreconditionNotMet);
   void delete_contract(
      in Contract a_contract )
      raises (PreconditionNotMet);
};


/************************************************************************
 * Cache : Manager of a set of related objects
 *     is associated to one DDS::Publisher and/or one DDS::Subscriber
 ************************************************************************/

local interface Cache : CacheBase {

   // Attributes
```

```
// ----------
readonly attribute DCPSState          pubsub_state;
readonly attribute DDS::Publisher     the_publisher;
readonly attribute DDS::Subscriber    the_subscriber;
readonly attribute boolean            updates_enabled;
readonly attribute ObjectHomeSeq      homes;
readonly attribute CacheAccessSeq     sub_accesses;
readonly attribute CacheListenerSeq   listeners;

// Operations
// ----------

//-- Infrastructure management
void register_all_for_pubsub()
   raises (
      BadHomeDefinition,
      DCPSError,
      PreconditionNotMet);
void enable_all_for_pubsub()
   raises (
      DCPSError,
      PreconditionNotMet);

// -- Home management
unsigned long register_home (
   in ObjectHome  a_home)
   raises (
      PreconditionNotMet);
ObjectHome find_home_by_name (
   in ClassName class_name);
ObjectHome find_home_by_index (
   in unsigned long index);

// -- Listener Management
void attach_listener (
   in CacheListener listener);
void detach_listener (
   in CacheListener listener);

// --- Updates management
void enable_updates ();
void disable_updates ();

// --- CacheAccess Management
CacheAccess create_access (
   in CacheUsage purpose)
   raises (
      PreconditionNotMet);
void delete_access (
   in CacheAccess access)
   raises (
```

```
            PreconditionNotMet);
    };

/***********************************************
 * CacheFactory : Factory to create Cache objects
 ***********************************************/

valuetype CacheDescription {
    public CacheName              name;
    public DDS::DomainParticipant domain;
    };

local interface CacheFactory {
    Cache create_cache (
        in CacheUsage      cache_usage,
        in CacheDescription cache_description)
        raises (
            DCPSError,
            AlreadyExisting);
    Cache find_cache_by_name(
        in CacheName       name);
    void delete_cache (
        in Cache           a_cache);
    };

};
```

2.2.1.2.2 Implied IDL

This section contains the implied IDL constructs for an application-defined class named ***Foo***.

```
#include "dds_dlrl.idl"

valuetype Foo: DDS::ObjectRoot {
    // some attributes and methods
    };

/***********************************
 * DERIVED CLASSES FOR Foo
 ***********************************/

typedef sequence<Foo> FooSeq;

local interface FooListener: DDS::ObjectListener {
    void on_object_created(
        in Foo the_object);
    void on_object_modified (
        in Foo the_object);
```

```
    void on_object_deleted(
       in Foo the_object);
    };
typedef sequence <FooListener> FooListenerSeq;

local interface FooSelectionListener : DDS::SelectionListener {
    void on_object_in (
       in Foo the_object);
    void on_object_modified (
       in Foo the_object);
    void on_object_out (
       in Foo the_object);

    };

 local interface FooFilter: DDS::FilterCriterion {
    boolean check_object (
       in Foo an_object,
       in DDS::MembershipState membership_state);
    };

local interface FooQuery : DDS::QueryCriterion, FooFilter {
    };


local interface FooSelection : DDS::Selection {
    readonly attribute FooFilter       filter;
    readonly attribute FooSeq          members;
    readonly attribute FooSelectionListener listener;

    FooSelectionListener set_listener (
       in FooSelectionListener listener);
    };
typedef sequence <FooSelection> FooSelectionSeq;

local interface FooHome : DDS::ObjectHome {
    readonly attribute FooSelectionSeqselections;
    readonly attribute FooListenerSeq  listeners;

    void attach_listener (
       in FooListener listener,
       in boolean concerns_contained_objects);
    void detach_listener (
       in FooListener listener);

    FooSelection create_selection (
       in FooFilter filter,
       in boolean auto_refresh)
       raises (
          DDS::BadParameter);
```

```
    void delete_selection (
        in FooSelection a_selection)
        raises (
            DDS::PreconditionNotMet);
    Foo create_object(
        in DDS::CacheAccess access)
        raises (
            DDS::PreconditionNotMet);
    Foo create_unregistered_object (
        in DDS::CacheAccess access)
        raises (
            DDS::PreconditionNotMet);
    void register_object (
        in Foo   unregistered_object)
        raises (
            DDS::AlreadyExisting,
            DDS::PreconditionNotMet);
    Foo find_object_in_access (
        in DDS::DLRLOid  oid,
        in DDS::CacheAccess access)
        raises (
            DDS::NotFound);
    Foo find_object (
        in DDS::DLRLOid  oid);
    FooSeq get_objects(
        in CacheBase source );
    FooSeq get_created_objects(
        in CacheBase source );
    FooSeq get_modified_objects(
        in CacheBase source );
    FooSeq get_deleted_objects(
        in CacheBase source );


    };


/****************************************
 * Derived class for relations to Foo
 ****************************************/
valuetype FooList : DDS::List {       //List<Foo>
    readonly attribute FooSeq values;
    void add( in Foo value );
    void put( in long key, in Foo value );
    Foo get( in long key );
};

valuetype FooSet : DDS::Set {                 // Set<Foo>
    FooSeq values ( );
    FooSeq added_elements( );
    FooSeq removed_elements( );
```

```
    boolean contains( in Foo value );
    void add( in Foo value );
    void remove( in Foo value );
};

valuetype FooStrMap : DDS::StrMap {       //StrMap<Foo>
    readonly attribute FooSeq values;
    void put( in string key, in Foo value );
    Foo get( in string key );
};

valuetype FooIntMap : DDS::IntMap {       //IntMap<Foo>
    readonly attribute FooSeq values;
    void put( in long key, in Foo value );
    Foo get( in long key );
};
```

2.2.2 Generation Process

2.2.2.1 Principles

The generic generation process explained in Section 2.1.4.6, “How is this Mapping Indicated?,” on page 2-11, is instantiated as follows:

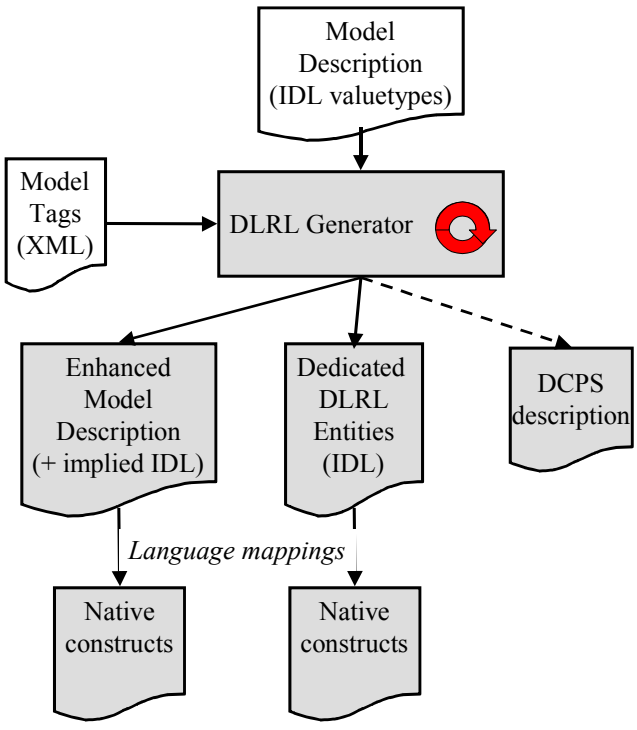


Figure 2-7 DLRL Generation Process (OMG IDL)

2.2.2.2 Model Description

As stated in Section 2.2.1, “Run-time Entities,” on page 2-45, application classes are modeled by means of IDL value-types.

Support for specific typed collections is introduced by means of a void value declaration, which will be transformed in the generation process by means of special model tags that are explained in the following section.

2.2.2.3 Model Tags

Model tags are specified by means of XML declarations that must be compliant with the DTD listed in the following section; subsequent sections give details on the constructs.

2.2.2.3.1 Model Tags DTD

The following is the DTD for expressing the Model Tags in XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT Dlrl
     (enumDef | templateDef | associationDef | compoRelationDef| classMapping)*>
<!ATTLIST Dlrl name CDATA #IMPLIED>

<!ELEMENT enumDef (value)*>
<!ATTLIST enumDef name CDATA  #REQUIRED>
<!ELEMENT value (#PCDATA)>

<!ELEMENT templateDef EMPTY>
<!ATTLIST templateDef name CDATA  #REQUIRED
                  pattern (List | StrMap | IntMap | Set) #REQUIRED
                  itemType CDATA  #REQUIRED>

<!ELEMENT associationDef (relation,relation)>
<!ELEMENT relation EMPTY>
<!ATTLIST relation  class CDATA  #REQUIRED
                attribute CDATA  #REQUIRED>

<!ELEMENT compoRelationDef EMPTY>
<!ATTLIST compoRelationDef class CDATA #REQUIRED
                attribute CDATA  #REQUIRED>

<!ELEMENT classMapping (mainTopic?,extensionTopic?,
(monoAttribute | multiAttribute | monoRelation | multiRelation | local)*)>
<!ATTLIST classMapping name CDATA  #REQUIRED>

<!ELEMENT mainTopic (keyDescription)>
<!ATTLIST mainTopic name CDATA  #REQUIRED
                    typename CDATA #IMPLIED>

<!ELEMENT extensionTopic (keyDescription)>
<!ATTLIST extensionTopic name CDATA  #REQUIRED
                    typename CDATA #IMPLIED>
```

```
<!ELEMENT monoAttribute (placeTopic?,valueField+)>
<!ATTLIST monoAttribute name CDATA  #REQUIRED>

<!ELEMENT multiAttribute (multiPlaceTopic,valueField+)>
<!ATTLIST multiAttribute name CDATA  #REQUIRED>

<!ELEMENT monoRelation (placeTopic?,keyDescription)>
<!ATTLIST monoRelation name CDATA  #REQUIRED>

<!ELEMENT multiRelation (multiPlaceTopic,keyDescription)>
<!ATTLIST multiRelation name CDATA  #REQUIRED>

<!ELEMENT local EMPTY>
<!ATTLIST local name CDATA  #REQUIRED>

<!ELEMENT placeTopic (keyDescription)>
<!ATTLIST placeTopic name CDATA  #REQUIRED
                     typename CDATA #IMPLIED>

<!ELEMENT multiPlaceTopic (keyDescription)>
<!ATTLIST multiPlaceTopic name CDATA  #REQUIRED
                     typename CDATA #IMPLIED
                     indexField CDATA  #IMPLIED>

<!ELEMENT keyDescription (keyField*)>
<!ATTLIST keyDescription content (FullOid | SimpleOid | NoOid) #REQUIRED>

<!ELEMENT keyField (#PCDATA)>

<!ELEMENT valueField (#PCDATA)>
```

2.2.2.3.2 Details on the XML constructs

To allow a better understanding, in the following examples, the DCPS information (topics, fields) is in capital letters, while the DLRL one is not.

2.2.2.3.2.1 Root

A DLRL Model Tags XML document, is a list of following XML tags:

- ***enumDef*** - to give explicit names to enumeration items, in case the default behavior (coding them by means of long values) is not suitable.
- ***templateDef*** - to define a typed collection or a reference (giving its pattern as well as the type of its elements; it comes in place of a statement such as List<Foo> which is not allowed in IDL.
- ***compoRelationDef*** - to state that a given relation is actually a composition.
- ***associationDef*** - to associate two relations, so that they make a full association (in the UML sense).
- ***classMapping*** - to define the mapping of a DLRL class to DCPS topics; it comprises a list of:
  - ***monoAttribute*** - for mono-valued attributes

- ***multiAttribute*** - for multi-valued attributes
- ***monoRelation*** - for mono-valued relations
- ***multiRelation*** - for multi-valued relations
- ***local*** - to state that an attribute is not a DLRL attribute (and thus will not be considered by this generation process).

2.2.2.3.2.2 EnumDef

This tag contains an attribute ***name*** (scoped name of the IDL enumeration) and as many ***value*** sub-tags that needed to give values.

Example:

```
<enumDef name="WeekDays">
  <value>Monday</value>
  <value>Tuesday</value>
  <value>Wednesday</value>
  <value>Thursday</value>
  <value>Friday</value>
  <value>Saturday</value>
  <value>Sunday</value>
</enumDef>
```

2.2.2.3.2.3 TemplateDef

This tag contains three attributes:

- ***name*** - gives the scoped name of the type.
- ***pattern*** - gives the construct pattern. The supported constructs are: ***List***, ***StrMap***, ***IntMap,*** and ***Set***.
- ***itemType*** - gives the type of each element in the collection.

Example:

```
<templateDef name="BarStrMap" pattern="StrMap" itemType="Bar"/>
```

This corresponds to a hypothetical `typedef StrMap<Foo> FooStrMap;`

2.2.2.3.2.4 AssociationDef

This tag puts in association two relations (that represent then the association ends of that association). It embeds two mandatory ***relation*** sub-tags to designate the concerned relations. Each of these sub-tags has two mandatory attributes:

- ***class*** - contains the scoped name of the class.
- ***attribute*** - contains the name of the attribute that supports the relation inside the class.

Example:

```
<associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
  </associationDef>
```

2.2.2.3.2.5 compoRelationDef

This tag states that the relation is actually a composition. It has two mandatory attributes:

- ***class*** - contains the scoped name of the class.
- ***attribute*** - contains the name of the attribute that supports the relation inside the class.

Example:

```
<compoRelationDef class"Radar" attribute="tracks"/>
```

2.2.2.3.2.6 ClassMapping

This tag contains one attribute ***name*** that gives the scoped name of the class and:

- an optional sub-tag ***mainTopic***;
- an optional sub-tag ***extensionTopic***;
- a list of attribute and/or relation descriptions.

Example:

```
<classMapping name="Track">
            ...
  </classMapping>
```

2.2.2.3.2.7 MainTopic

This tag gives the main DCPS ***Topic***, to which that class refer. The main ***Topic*** is the topic that gives the existence of a object (an object is declared as existing if, and only if, there is an instance in that ***Topic*** matching its ***key*** value.

It comprises one attribute (***name)*** that gives the name of the ***Topic***, one (optional) attribute (***typename)*** that gives the name of the type (if this attribute is not supplied the type name is considered to be equal to the topic name) and:

- a mandatory sub-tag ***keyDescription***.

Example:

```
<mainTopic name="TRACK-TOPIC" typename=”TrackType”>
            <keyDescription
            ...
            </keyDescription>
</mainTopic>
```

2.2.2.3.2.8 KeyDescription

This tag describes the key to be associated to several elements (***mainTopic***, ***extensionTopic***, ***placeTopic***, and ***multiPlaceTopic***).

It comprises an attribute that describes the content of the ***keyDescription***, that can be:

- ***FullOid***, in that case, the key description should contain as first ***keyField*** the name of the ***Topic*** field used to store the class name and as second ***keyField*** the name of the ***Topic*** field used to store the OID itself.

- ***SimpleOid***, in that case the key description should only contain one ***keyField*** to contain the OID itself.
- ***NoOid***, in that case the case description should contain as many ***keyField*** that are needed to identify uniquely one row in the related ***Topic*** and it is the responsibility of the DLRL implementation to manage the association between those fields and the ***DLRLOid*** as perceived by the application developer.

It contains also as many elements ***keyField*** as needed.

Example:

```
<keyDescription content="SimpleOid">
        <keyField>OID</keyField>
</keyDescription>
```

2.2.2.3.2.9 ExtensionTable

This tag gives the DCPS ***Topic*** that is used as an extension table for the attributes. It comprises the same attributes as ***mainTopic***.

2.2.2.3.2.10 MonoAttribute

This tag gives the mapping for a mono-valued attribute. It has :

- A mandatory attribute to give the ***name*** of the attribute.
- An optional sub-tag to give the DCPS ***Topic*** where it is placed (***placeTopic***). This sub-tag follows the same pattern as ***mainTopic***. In case it is not given, the ***extensionTopic***, or if there is no ***extensionTopic***, the ***mainTopic*** is used in place of ***placeTopic***.
- One or more ***valueField*** sub-tag(s) to give the name of the field(s) that will contain the value of that attribute.

Example:

```
<monoAttribute name="y">
        <placeTopic name="Y_TOPIC">
                <keyDescription content="SimpleOID">
                        <keyField>OID</keyField>
                </keyDescription>
        </placeTopic>
        <valueField>Y</valueField>
</monoAttribute>
```

2.2.2.3.2.11 MultiAttribute

This tag gives the mapping for a multi-valued attribute. It has:

- A mandatory attribute to give the ***name*** of the attribute.
- A mandatory sub-tag to give the DCPS ***Topic*** where it is placed (***multiPlaceTopic***). This sub-tag follows the same pattern as ***placeTopic***, except it has a mandatory attribute in addition to state the field needed for storing the collection index.

- One or more ***valueField*** sub-tag(s) to give the name of the field(s) that will contain the value of that attribute.

Example:

```
<multiAttribute name="comments">
        <multiPlaceTopic name="COMMENTS-TOPIC"
                <keyDescription content="FullOID">
                        <keyField>CLASS</keyField>
                        <keyField>OID</keyField>
                </keyDescription>
        </multiPlaceTopic>
        <valueField>COMMENT</valueField>
</multiAttribute>
```

2.2.2.3.2.12 MonoRelation

This tag gives the mapping for a mono-valued attribute. It has:

- A mandatory attribute to give the ***name*** of the attribute.
- An optional sub-tag to give the ***Topic*** where it is placed (***placeTopic*** – see Section 2.2.2.3.2.10, “MonoAttribute).
- One ***keyDescription*** sub-tag to give the name of the field(s) that will contain the value of that relation (i.e., a place holder to a reference to the pointed object).

Example:

```
<monoRelation name="a_radar">
        <keyDescription content="SimpleOID">
                <keyField>RADAR_OID</keyField>
        </keyDescription>
</monoRelation>
```

2.2.2.3.2.13 MultiRelation

This tag gives the mapping for a multi-valued relation. It has:

- A mandatory attribute to give the ***name*** of the relation.
- A mandatory sub-tag to give the DCPS ***Topic*** where it is placed (***multiPlaceTopic*** – see Section 2.2.2.3.2.11).
- One ***valueKey*** sub-tag (see Section 2.2.2.3.2.12).

Example:

```
<multiRelation name="tracks">
        <multiPlaceTopic name="RADARTRACKS-TOPIC"
                <keyDescription content="SimpleOID">
                        <keyField>RADAR-OID</keyField>
                </keyDescription>
        <\multiPlaceTopic>
        <keyDescription content="FullSimpleOID">
                <keyField>TRACK-CLASS</keyField>
                <keyField>TRACK-OID</keyField>
        </keyDescription>
```

```
</multiRelation>
```

2.2.2.3.2.14 Local

This tag just indicates that the corresponding attribute (designated by its name) has to be ignored by the service.

Example:

```
<local name="w"/>
```

2.2.3 Example

This section contains a very simple example, to illustrate DLRL.

2.2.3.1 UML Model

The following UML diagram describes a very simple application model with three classes:

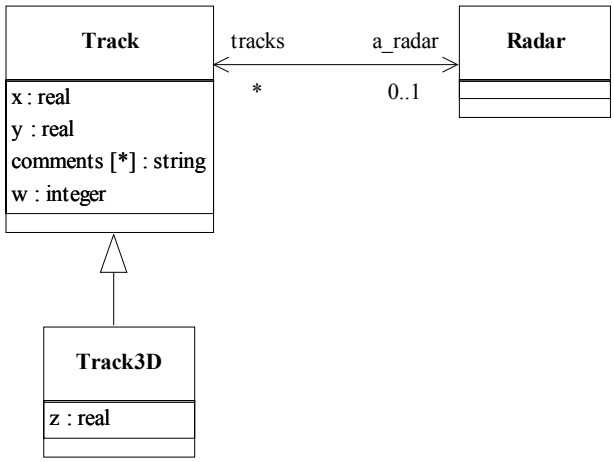


Figure 2-8 UML Class Diagram of the Example

2.2.3.2 IDL Model Description

Based on this model, the model description (IDL provided by the application developer) could be:

```
#include "dlrl.idl"

valuetype stringStrMap;     // StrMap<string>
valuetype TrackList;        // List<Track>
valuetype Radar;

valuetype Track : DLRL::ObjectRoot {
```

```
    public double      x;
    public double      y;
    public stringStrMap comments;
    public long        w;
    public Radar a_radar;
    };

valuetype Track3D : Track {
    public double      z;
    };

valuetype Radar : DLRL::ObjectRoot {
    public TrackList    tracks;
    };
```

2.2.3.3 XML Model Tags

The following UML tags to drive the generation process could then be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Dlrl SYSTEM "dlrl.dtd">
<Dlrl name="example">
  <templateDef name="StringStrMap" pattern="StrMap" itemType="string"/>
  <templateDef name="TrackList" pattern="List" itemType="Track"/>
  <classMapping name="Track">
    <mainTopic name="TRACK-TOPIC">
      <keyDescription content="FullOid">
        <keyField>CLASS</keyField>
        <keyField>OID</keyField>
      </keyDescription>
    </mainTopic>
    <monoAttribute name="x">
      <valueField>X</valueField>
    </monoAttribute>
    <monoAttribute name="y">
      <placeTopic name="Y_TOPIC">
        <keyDescription content="FullOid">
          <keyField>CLASS</keyField>
          <keyField>OID</keyField>
        </keyDescription>
      </placeTopic>
      <valueField>Y</valueField>
    </monoAttribute>
    <multiAttribute name="comments">
      <multiPlaceTopic name="COMMENTS-TOPIC" indexField="INDEX">
        <keyDescription content="FullOid">
          <keyField>CLASS</keyField>
          <keyField>OID</keyField>
        </keyDescription>
      </multiPlaceTopic>
      <valueField>COMMENT</valueField>
    </multiAttribute>
```

```
    <monoRelation name="a_radar">
      <keyDescription content="SimpleOid">
        <keyField>RADAR_OID</keyField>
      </keyDescription>
    </monoRelation>
    <local name="w"/>
  </classMapping>
  <classMapping name="Track3D">
    <mainTopic name="TRACK-TOPIC">
      <keyDescription content="FullOid">
        <keyField>CLASS</keyField>
        <keyField>OID</keyField>
      </keyDescription>
    </mainTopic>
    <extensionTopic name="TRACK3D-TOPIC">
      <keyDescription content="FullOid">
        <keyField>CLASS</keyField>
        <keyField>OID</keyField>
      </keyDescription>
    </extensionTopic>
    <monoAttribute name="z">
      <valueField>Z</valueField>
    </monoAttribute>
  </classMapping>
  <classMapping name="Radar">
    <mainTopic name="RADAR-TOPIC">
      <keyDescription content="SimpleOid">
        <keyField>OID</keyField>
      </keyDescription>
    </mainTopic>
    <multiRelation name="tracks">
      <multiPlaceTopic name="RADARTRACKS-TOPIC" indexField="INDEX">
        <keyDescription content="SimpleOid">
          <keyField>RADAR-OID</keyField>
        </keyDescription>
      </multiPlaceTopic>
      <keyDescription content="FullOid">
        <keyField>TRACK-CLASS</keyField>
        <keyField>TRACK-OID</keyField>
      </keyDescription>
    </multiRelation>
  </classMapping>
  <associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
  </associationDef>
</Dlrl>
```

It should be noted that XML is not suitable for manual editing, therefore the file seems much more complicated than it actually is. It seems much simpler when viewed through an XML editor, as the following picture illustrates.

:

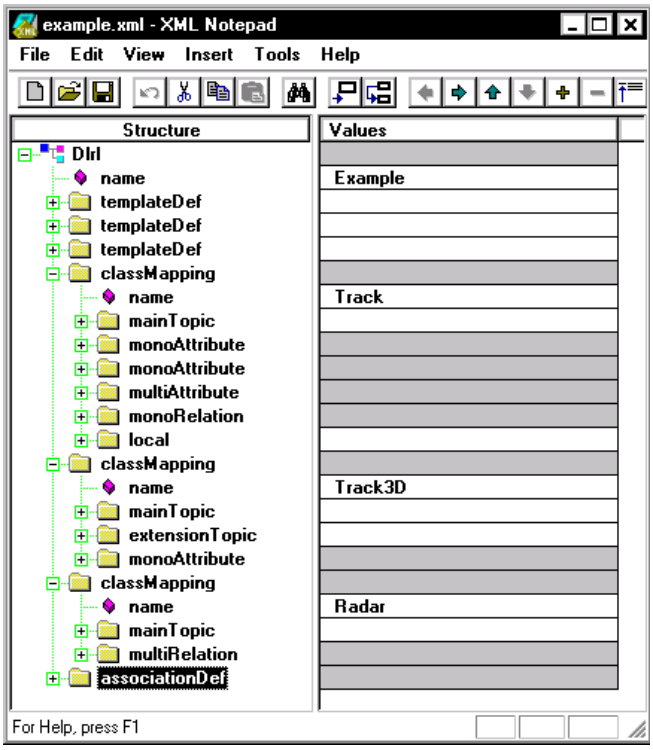


Figure 2-9 XML Editor Illustration

Also note that only the three ***templateDef***, the ***associationDef***, and the ***local***[16] tags are mandatory in all cases. The ***ClassMapping*** tags are only required if a deviation is wanted from the default mapping described in Section 2.1.4.3, “Default Mapping,” on page 2-8. In case no deviation is wanted from the default mapping, the XML description can be restricted to the following minimum:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Dlrl SYSTEM "dlrl.dtd">
<Dlrl name="Example">
  <templateDef name="stringStrMap" pattern="StrMap" itemType="string"/>
  <templateDef name="TrackList" pattern="List" itemType="Track"/>
  <classMapping name="Track">
<local name="w"/>
  </classMapping>
  <associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
  </associationDef>
</Dlrl>
```

16.To state that Track::w is not a DLRL attribute.

A following step could be to define UML 'tags'[17] and to generate those files based on the UML model. However, this is far beyond the scope of this specification.

2.2.3.4 Underlying DCPS Data Model

This mapping description assumes that the underlying DCPS data model is made of five topics with their fields as described in the following tables:

TRACK-TOPIC	Topic to store all ***Track*** objects (including the derived classes) – as well as the embedded attributes/relations defined on ***Track***.
CLASS	Field to store the ***class*** part of the object reference.
OID	Field to store the ***oid*** part of the object reference.
X	Field to store the value of the attribute ***x***.
RADAR-OID	Field to store the relation ***a_radar***.

Y-TOPIC	Topic to store ***Track::y,*** outside ***Track***'s main topic.
CLASS	Field to store the ***class*** part of the object reference.
OID	Field to store the ***oid*** part of the object reference.
Y	Field to store the value of the attribute y.

COMMENTS-TOPIC	Topic to store ***Track::comments*** (required as it is a collection).
CLASS	Field to store the ***class*** part of the owning object reference (here a ***Track***).
OID	Field to store the ***oid*** part of the owning object reference (here a ***Track***).
INDEX	Field to store the ***index*** part in the collection
COMMENT	Field to store one element of the attribute ***comments***.

17. This specification does not address this point and therefore does not say anything about how this should/could be represented in UML. The interface between the modeling phase and the coding phase has just been designed as simple as possible, so that it would be very easy to fill the gap.

TRACK3D-TOPIC	Topic to store the embedded attributes/relations added on ***Track3D*** (here only *z*).
CLASS	Field to store the ***class*** part of the object reference.
OID	Field to store the ***oid*** part of the object reference.
Z	Field to store the value of the attribute *z*.

RADARTRACKS-TOPIC	Topic to store ***Radar::tracks*** (required as it is a collection).
RADAR-OID	Field to store the reference to the owning object (here a ***Radar***).
INDEX	Field to store ***index*** in the collection.
TRACK-CLASS	Field to store the ***class*** part of a reference to an item in the collection (here a ***Track***).
TRACK-OID	Field to store the ***oid*** part of a reference to an item in the collection (here a ***Track***).

Note that references to ***Track*** objects (including derived ***Track3D***) must provision a field for the class indication, while references to ***Radar*** objects do not, for the ***Radar*** class has no subclasses and does not share its main Topic.

2.2.3.5 Code Example

The following text is a very simple, non fully running, C++ example just to give the flavor of how objects can be created, modified, and then published.

```
DDS::DomainParticipant_var dp;
DLRL::CacheFactory_var cf;

/*
 * Init phase
 */
DLRL::Cache_var c = cf->create_cache (WRITE_ONLY, dp);
RadarHome_var rh;
TrackHome_var th;
Track3DHome_var t3dh;

c->register_home (rh);
c->register_home (th);
c->register_home (t3dh);
c->register_all_for_pubsub();
// some QoS settings if needed
c->enable_all_for_pubsub();
```

```
/*
 * Creation, modifications and publication
 */
Radar_var r1 = rh->create_object(c);
Track_var t1 = th->create-object (c);
Track3D_var t2 = t3dh->create-object (c);
t1->w(12);// setting of a pure local attribute
t1->x(1000.0);// some DLRL attributes settings
t1->y(2000.0);
t2->a_radar->put(r1);// modifies r1->tracks accordingly
t2->x(1000.0);
t2->y(2000.0);
t2->z(3000.0);
t2->a_radar->put(r1);// modifies r1->tracks accordingly
c->write();// all modifications are published
};
```

Compliance Points *A*

This specification has a single mandatory compliance profile, which includes the complete specification.

Compliance with the DLRL specification is equivalent to complying with the “Object Model Profile” of the Data Distribution Service Specification version 1.2.

Syntax for DLRL Queries and Filters C

The syntax, defined with the BNF-grammar below, is used to express a filter or a query expression in the DLRL constructs:

- The ***filter*** in the ***FilterCriterion*** (see Section 2.1.6.3.11, “FilterCriterion,” on page 2-32” on page 4-23).
- The ***query*** in the ***QueryCriterion*** (see Section 2.1.6.3.12, “QueryCriterion,” on page 2-32” on page 4-27).

The following notational conventions are made:

- The *NonTerminals* are typeset in italics.
- The `'Terminals'` are quoted and typeset in a fixed width font.
- The TOKENS are typeset in small caps.
- The notation *(element // ',')* represents a non-empty comma-separated list of *elements*.

Query grammar in BNF

```
                .
Condition       ::= Predicate
                |   Condition 'AND' Condition
                |   Condition 'OR' Condition
                |   'NOT' Condition
                |   '(' Condition ')'
                .
Predicate       ::= ComparisonPredicate
                |   BetweenPredicate
                .
ComparisonPredicate  ::=  FIELDNAME RelOp Parameter
                     |    Parameter RelOp FIELDNAME
                     |    FIELDNAME RelOp FIELDNAME
                     .
BetweenPredicate          ::=    FIELDNAME 'BETWEEN' Range
                     |    FIELDNAME 'NOT BETWEEN' Range
                     .
RelOp           ::= '=' | '>' | '>=' | '<' | '<=' | '<>'
```

```
                .
Range           ::= Parameter 'AND' Parameter
                .
Parameter       ::= INTEGERVALUE
                |   CHARVALUE
                |   FLOATVALUE
                |   STRING
                |   ENUMERATEDVALUE
                |   PARAMETER
                .
```

Token expression

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- FIELDNAME - A fieldname is a reference to a field in the data-structure. The dot `'.'` is used to navigate through nested structures. The number of dots that may be used in a FIELD-NAME is unlimited. The `'[INTEGERVALUE|STRING]'` construct is used to navigate in a collection. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the field-names that appear on the language-specific (e.g., C/C++, Java) mapping of the structure.
- INTEGERVALUE - Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by `0x` and must be a valid hexadecimal expression.
- CHARVALUE - A single character enclosed between single quotes.
- FLOATVALUE - Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point (`'.'`). A power-of-ten expression may be postfixed, which has the syntax e*n*, where *n* is a number, optionally preceded by a plus or minus sign.
- STRING - Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote.
- ENUMERATEDVALUE - An enumerated value is a reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.
- PARAMETER - A parameter is of the form %*n*, where *n* represents a natural number (zero included) smaller than 100. It refers to the $n + 1^{th}$ argument in the given context.

C
Cache 22
CacheAccess 20
CacheFactory 19
CacheListener 25
Code Example 74
Collection 38
compliance 1
CORBA
  contributors v

D
Data Local Reconstruction Layer (DLRL) 3
data model 3
data-centric exchange 2
Data-Centric Publish-Subscribe (DCPS) model 3
DCPS 2
DCPS (Data-Centric Publish-Subscribe) 1
DCPS data model 73
DLRL 1
DLRL (Data Local Reconstruction Layer) 1
DLRL metamodel 4
DLRL objects 2

G
generation process 62
generation tool 2

I
IDL Model Description 69
IntMap 41

L
List 38

M
mapping of an object reference 7
mapping of attributes and relations 7
mapping of classes 6
mapping when DCPS model is fixed 11
mapping, default 8
mapping, operational 13
Model Tags DTD 63

O
ObjectHome 26
ObjectListener 29
ObjectRoot 33

P
Platform Specific Model (PSM) 45

Q
QoS (Quality of Service) 2

R
run-time entities 45

S
Security Service 1
Selection 30
SelectionListener 33
StrMap 40

syntax for DLRL queries and filters 1

T
typed interfaces 2

U
UML Model 69

X
XML Model Tags 70