Data Distribution Service for Real-Time Systems Specification

This OMG document replaces the submission document (mars/03-03-05) and draft adopted specification (ptc/03-04-03). It is an OMG Final Adopted Specification and is currently in the finalization phase. (This document represents an editorial update and supersedes ptc/03-05-20.) Comments on the content of this document are welcomed, and should be directed to issues@omg.org by November 24, 2003.

You may view the pending issues for this specification from the OMG revision issues web page http://cgi.omg.org/issues/; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on February 29, 2004. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.
Data Distribution Service for Real-Time Systems Specification

May 2003
Final Adopted Specification
ptc/03-07-07

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION,
INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube Logo™, MOO™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designsations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.
## Contents

1. **Overview** ................................................................. 1-1  
   1.1 Introduction ......................................................... 1-1  
   1.2 Purpose ............................................................ 1-2  
2. **Data-Centric Publish-Subscribe (DCPS)** ................. 2-1  
   2.1 Platform Independent Model (PIM) ............................. 2-2  
      2.1.1 Overview and Design Rationale ......................... 2-2  
      2.1.2 PIM Description ....................................... 2-9  
      2.1.3 Supported QoS ........................................ 2-60  
      2.1.4 Listeners, Conditions and Wait-sets ............... 2-75  
      2.1.5 Built-in Topics ...................................... 2-88  
      2.1.6 Interaction Model .................................... 2-90  
   2.2 CORBA Platform Specific Model (PSM) ..................... 2-97  
      2.2.1 Introduction ............................................ 2-97  
      2.2.2 PIM to PSM Mapping Rules ......................... 2-97  
      2.2.3 DCPS PSM : IDL ...................................... 2-98  
3. **Data Local Reconstruction Layer (DLRL)** .............. 3-1  
   3.1 PIM ..................................................................... 3-1  
      3.1.1 Overview and Design Rationale .................... 3-1  
      3.1.2 DLRL Description ...................................... 3-2  
      3.1.3 What Can Be Modeled with DLRL ................ 3-2  
      3.1.4 Structural Mapping .................................. 3-6  
      3.1.5 Operational Mapping .................................. 3-13  
      3.1.6 Functional Mapping .................................. 3-14  
   3.2 CORBA PSM ...................................................... 3-40
3.2.1 CORBA Run-time Entities .......................... 3-41
3.2.2 Generation Process ................................. 3-55
3.2.3 Example .............................................. 3-60

Appendix A - Compliance Points ......................... A-1
Appendix B - Syntax for DCPS Queries and Filters .... B-1
Appendix C - Syntax for DLRL Queries and Filters ..... C-1
Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization’s charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG’s objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.
The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry’s premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/.

**OMG Documents**

The OMG Specifications Catalog is available from the OMG website at:


The OMG documentation is organized as follows:

**OMG Modeling Specifications**
Includes the UML, MOF, XMI, and CWM specifications.

**OMG Middleware Specifications**
Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

**Platform Specific Model and Interface Specifications**
Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.
Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the Object Management Architecture Guide.) OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.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

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction”</td>
<td>1-1</td>
</tr>
<tr>
<td>“Purpose”</td>
<td>1-2</td>
</tr>
</tbody>
</table>

1.1 Introduction

This specification describes two levels of interfaces:

- A lower DCPS (Data-Centric Publish-Subscribe) level that is targeted towards the efficient delivery of the proper information to the proper recipients.
- An optional higher DLRL (Data Local Reconstruction Layer) level, which allows for a simple integration of the Service into the application layer.

The expected application domains require DCPS to be high-performance and predictable as well as efficient in its use of resources. To meet these requirements it is important that the interfaces are designed in such a way that they:

- allow the middleware to pre-allocate resources so that dynamic resource allocation can be reduced to the minimum,
- avoid properties that may require the use of unbounded or hard-to-predict resources,
- and
- minimize the need to make copies of the data.
Even at the DCPS level, typed interfaces (i.e., interfaces that take into account the actual data types) are preferred to the extent possible. Typed interfaces 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.

QoS (Quality of Service) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer only indicates ‘what’ is wanted rather than ‘how’ this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is then an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to more flexibility.

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

Many real-time applications have a requirement to model some of their communication patterns as a pure data-centric exchange, where applications publish (supply or stream) “data” which is then available to the remote applications that are interested in it. Relevant real-time applications can be found in C4I, industrial automation, distributed control and simulation, telecom equipment control, sensor networks, and network management systems. More generally, any application requiring (selective) information dissemination is a candidate for a data-driven network architecture.

Predictable distribution of data with minimal overhead is of primary concern to these real-time applications. Since it is not feasible to infinitely extend the needed resources, it is important to be able to specify the available resources and provide policies that allow the middleware to align the resources to the most critical requirements. This necessity translates into the ability to control Quality of Service (QoS) properties that affect predictability, overhead, and resource utilization.

The need to scale to hundreds or thousands of publishers and subscribers in a robust manner is also an important requirement. This is actually not only a requirement of scalability but also a requirement of flexibility: on many of these systems, applications
are added with no need/possibility to reconstruct the whole system. Data-centric communications decouples senders from receivers; the less coupled the publishers and the subscribers are, the easier these extensions become.

Distributed shared memory is a classic model that provides data-centric exchanges. However, this model is difficult to implement efficiently over a network and does not offer the required scalability and flexibility. Therefore, another model, the **Data-Centric Publish-Subscribe (DCPS)** model, has become popular in many real-time applications. This model builds on the concept of a “global data space” that is accessible to all interested applications. Applications that want to contribute information to this data space declare their intent to become “Publishers.” Similarly, applications that want to access portions of this data space declare their intent to become “Subscribers.” Each time a Publisher posts new data into this “global data space,” the middleware propagates the information to all interested Subscribers.

Underlying any data-centric publish subscribe system is a **data model**. This model defines the “global data space” and specifies how Publishers and Subscribers refer to portions of this space. The data-model can be as simple as a set of unrelated data-structures, each identified by a **topic** and a **type**. The topic provides an identifier that uniquely identifies some data items within the global data space. The type provides structural information needed to tell the middleware how to manipulate the data and also allows the middleware to provide a level of type safety. However, the target applications often require a higher-level data model that allows expression of aggregation and coherence relationships among data elements.

Another common need is a **Data Local Reconstruction Layer (DLRL)** that automatically reconstructs the data locally from the updates and allows the application to access the data ‘as if’ it were local. In that case, the middleware not only propagates the information to all interested subscribers but also updates a local copy of the information.

There are commercially-available products that implement DCPS fully and the DLRL partially (among them, NDDS from Real-Time Innovations and Splice from THALES Naval Nederland); however, these products are proprietary and do not offer standardized interfaces and behavior that would allow portability of the applications built upon them. **The purpose of this specification is to offer those standardized interfaces and behavior.**

---

1. In addition to topic and type, it is sometimes desirable for subscriptions to further refine the data they are interested in based on the content of the data itself. These so-called content-based subscriptions are gaining popularity in large-scale systems.
Data-Centric Publish-Subscribe (DCPS)

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Independent Model (PIM)</td>
<td>2-2</td>
</tr>
<tr>
<td>CORBA Platform Specific Model (PSM)</td>
<td>2-97</td>
</tr>
</tbody>
</table>

This chapter describes the mandatory DCPS layer. The DCPS layer provides the functionality required for an application to publish and subscribe to the values of data objects.

It allows:

• Publishing applications to identify the data objects they intend to publish, and then provide values for these objects.

• Subscribing applications to identify which data objects they are interested in, and then access their data values.

• Applications to define topics, to attach type information to the topics, to create publisher and subscriber entities, to attach QoS policies to all these entities and, in summary, to make all these entities operate.

The description is organized into two subsections:

• The Platform Independent Model (PIM)

• The Platform Specific Model (PSM) for the CORBA platform based on the PIM.
2.1 Platform Independent Model (PIM)

2.1.1 Overview and Design Rationale

2.1.1.1 Format and conventions

The purpose of this subsection is to provide an operational overview of the DCPS PIM. To do so, it introduces many terms. Some of them are common terms whose meaning, in the context of publish-subscribe, is different from common usage. In cases where it is deemed appropriate, such terms will be italicized. Other terms are unique to publish-subscribe and/or to this specification, and are incorporated as key elements of the Class Model. The first time such terms are used, they will be formatted with Bold-italics. Subsequent occurrences may not be highlighted in any way.

In addition to the UML diagrams, all the classes that constitute the Service are documented using tables. The format used to document these classes is shown below:

```
<class name>

attributes
<attribute name> <attribute type>
...

operations
<operation name> <return type>
<parameter> <parameter type>
...

```

The operation <parameter> can contain the modifier “in”, “out”, or “inout” ahead of the parameter name. If this modifier is omitted, it is implied that the parameter is an “in” parameter.

In some cases the operation parameters or return value(s) are a collection with elements of a given <type>. This is indicated with the notation “<type> [ ]”. This notation does not imply that it will be implemented as an array. The actual implementation is defined by the PSM: it may end up being mapped to a sequence, a list, or other kind of collection.

For example, the class named ‘MyClass’ below has a single attribute, named 'my_attribute' of type 'long' and a single operation 'my_operation' that returns a long. The operation takes four parameters. The first, ‘param1’, is an output parameter of type

---

1. In this case, the written name is exactly the one of the corresponding class, which forbids the use of the plural. In case this would lead to ambiguity, it has been followed by 'objects' to state that there may not be only one of these.
long; the second, ‘param2’, an input-output parameter of type long; the third, ‘param3’, is an input parameter (the “in” modifier is implied by omission) of type long; and the fourth, ‘param4’, is also an input parameter of type collection of longs\(^2\).

At the PIM level we have modeled errors as operation return codes typed \texttt{ReturnCode\_t}. Each PSM may map these to either return codes or exceptions. The complete list of return codes is indicated below.

<table>
<thead>
<tr>
<th>Return codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>Successful return.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Generic, unspecified error.</td>
</tr>
<tr>
<td>BAD_PARAMETER</td>
<td>Illegal parameter value.</td>
</tr>
<tr>
<td>UNSUPPORTED</td>
<td>Unsupported operation. Can only be returned by operations that are optional.</td>
</tr>
<tr>
<td>OUT_OF_RESOURCES</td>
<td>Service ran out of the resources needed to complete the operation.</td>
</tr>
<tr>
<td>NOT_ENABLED</td>
<td>Operation invoked on an \texttt{Entity} that is not yet enabled.</td>
</tr>
<tr>
<td>IMMUTABLE_POLICY</td>
<td>Application attempted to modify an immutable \texttt{QosPolicy}.</td>
</tr>
<tr>
<td>INCONSISTENT_POLICY</td>
<td>Application specified a set of policies that are not consistent with each other.</td>
</tr>
<tr>
<td>PRECONDITION_NOT_MET</td>
<td>A pre-condition for the operation was not met.</td>
</tr>
</tbody>
</table>

Any operation with return type \texttt{ReturnCode\_t} may return OK or ERROR. Any operation that takes an input parameter may additionally return BAD\_PARAMETER. OK, ERROR, and BAD\_PARAMETER are the standard return codes. Operations that may return any of the additional error codes above will state so explicitly.

---

2. That is, a collection where the type of each element is ‘long’.
2.1.1.2 Conceptual Outline

2.1.1.2.1 Overview

Information flows with the aid of the following constructs\(^3\): Publisher and DataWriter on the sending side, Subscriber, and DataReader on the receiving side.

- A Publisher is an object responsible for data distribution. It may publish data of different data types. A DataWriter acts as a typed\(^4\) accessor to a publisher. The DataWriter is the object the application must use to communicate to a publisher the existence and value of data-objects of a given type. When data-object values have been communicated to the publisher through the appropriate data-writer, it is the publisher's responsibility to perform the distribution (the publisher will do this according to its own QoS, or the QoS attached to the corresponding data-writer). A

---

3. All those constructs are local to the application part. Actually they play the role of proxies to the service.

4. 'typed' means that each DataWriter object is dedicated to one application data-type.
A **Subscriber** is an object responsible for receiving published data and making it available (according to the Subscriber's QoS) to the receiving application. It may receive and dispatch data of different specified types. To access the received data, the application must use a typed **DataReader** attached to the subscriber. Thus, a **subscription** is defined by the association of a data-reader with a subscriber. This association expresses the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber.

**Topic** objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A **Topic** is meant to fulfill that purpose: it associates a name (unique in the domain\(^5\)), a data-type, and QoS related to the data itself. In addition to the topic QoS, the QoS of the **DataWriter** associated with that **Topic** and the QoS of the **Publisher** associated to the **DataWriter** control the behavior on the publisher's side, while the corresponding **Topic**, **DataReader**, and **Subscriber** QoS control the behavior on the subscriber's side.

When an application wishes to publish data of a given type, it must create a **Publisher** (or reuse an already created one) and a **DataWriter** with all the characteristics of the desired publication. Similarly, when an application wishes to receive data, it must create a **Subscriber** (or reuse an already created one) and a **DataReader** to define the subscription.

### 2.1.1.2.2 Overall Conceptual Model

The overall conceptual model is shown in Figure 2-2 on page 2-7. Notice that all the main communication objects (the specializations of **Entity**) follow unified patterns of:

- **Supporting QoS** (made up of several **QosPolicy**); QoS provides a generic mechanism for the application to control the behavior of the Service and tailor it to its needs. Each **Entity** supports its own specialized kind of QoS policies. The complete list of QoS policies and their meaning is described in Section 2.1.3, "Supported QoS," on page 2-60.

- **Accepting a Listener**\(^6\); listeners provide a generic mechanism for the middleware to notify the application of relevant asynchronous events, such as arrival of data corresponding to a subscription, violation of a QoS setting, etc. Each DCPS entity

---

5. Broadly speaking, a domain represents the set of applications that are communicating with each other. This concept is defined more precisely in Overall Conceptual Model2.1.1.2.2 and DomainParticipant Class.

6. This specification made the choice of allowing the attachment of only one Listener per entity (instead of a list of them). The reason for that choice is that this allows a much simpler (and, thus, more efficient) implementation as far as the middleware is concerned. Moreover, if it were required, implementing a listener that, when triggered, triggers in return attached 'sub-listeners', can be easily done by the application.
supports its own specialized kind of listener. Listeners are related to changes in status conditions. This relationship is described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

- Accepting a StatusCondition (and a set of ReadCondition objects for the DataReader); conditions (in conjunction with WaitSet objects) provide support for an alternate communication style between the middleware and the application (i.e., wait-based rather than notification-based). The complete set of status conditions is described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

All these DCPS entities are attached to a DomainParticipant. A domain participant represents the local membership of the application in a domain. A domain is a distributed concept that links all the applications able to communicate with each other. It represents a communication plane: only the publishers and the subscribers attached to the same domain may interact.

DomainEntity is an intermediate object whose only purpose is to state that a DomainParticipant cannot contain other domain participants.
At the DCPS level, data types represent information that is sent atomically\(^7\). For performance reasons, only plain data structures are handled by this level.

By default, each data modification is propagated individually, independently, and uncorrelated with other modifications. However, an application may request that several modifications be sent as a whole and interpreted as such at the recipient side. This functionality is offered on a Publisher/Subscriber basis. That is, these relationships can only be specified among DataWriter objects attached to the same Publisher and retrieved among DataReader objects attached to the same Subscriber.

\(^7\) Note that the optional DLRL layer provides the means to break data-objects into separate elements, each sent atomically.
By definition, a **Topic** corresponds to a single data type. However, several topics may refer to the same data type. Therefore, a **Topic** identifies data of a single type, ranging from one single instance to a whole collection of instances of that given type. This is shown in Figure 2-3 for the hypothetical data-type “Foo.”

![Figure 2-3](image)

A topic can identify a collection of data-object instances

In case a set of instances is gathered under the same topic, different instances must be distinguishable. This is achieved by means of the values of some data fields that form the **key** to that data set. The **key** description (i.e., the list of data fields whose value forms the key) has to be indicated to the middleware. The rule is simple: *different data values with the same key value represent successive values for the same instance, while different data values with different key values represent different instances.* If no key is provided, the data set associated with the **Topic** is restricted to a single instance.

Topics need to be known by the middleware and potentially propagated. **Topic** objects are created using the create operations provided by **DomainParticipant**.

The interaction style is straightforward on the publisher’s side: when the application decides that it wants to make data available for publication, it calls the appropriate operation on the related **DataWriter** (this, in turn, will trigger its **Publisher**).

On the subscriber’s side however, there are more choices: relevant information may arrive when the application is busy doing something else or when the application is just waiting for that information. Therefore, depending on the way the application is designed, asynchronous notifications or synchronous access may be more appropriate. Both interaction modes are allowed, a **Listener** is used to provide a callback for synchronous access and a **WaitSet** associated with one or several **Condition** objects provides asynchronous data access.

The same synchronous and asynchronous interaction modes can also be used to access changes that affect the middleware communication status. For instance, this may occur when the middleware asynchronously detects an inconsistency. In addition, other
middleware information that may be relevant to the application (such as the list of the existing topics) is made available by means of built-in topics that the application can access as plain application data, using built-in\textsuperscript{8} data-readers.

### 2.1.2 PIM Description

The DCPS is comprised of five modules:

![DCPS module breakdown](image)

- **The Infrastructure Module** defines the abstract classes and the interfaces that are refined by the other modules. It also provides support for the two interaction styles (notification- and wait-based) with the middleware.

- **The Domain Module** contains the `DomainParticipant` class that acts as an entry-point of the Service and acts as a factory for many of the classes. The `DomainParticipant` also acts as a container for the other objects that make up the Service.

- **The Topic-Definition Module** contains the `Topic`, `ContentFilteredTopic`, and `MultiTopic` classes, the `TopicListener` interface, and more generally, all that is needed by the application to define `Topic` objects and attach QoS policies to them.

---

\textsuperscript{8} These built-in data-readers should be provided with every implementation of the service. They are further described in Section 2.1.5, “Built-in Topics,” on page 2-88.
The Publication Module contains the Publisher and DataWriter classes as well as the PublisherListener and DataWriterListener interfaces, and more generally, all that is needed on the publication side.

The Subscription Module contains the Subscriber, DataReader, ReadCondition, and QueryCondition classes, as well as the SubscriberListener and DataReaderListener interfaces, and more generally, all that is needed on the subscription side.

At the PIM level, we have chosen to model any entity as a class or interface. It should be noted, however, that this does not mean that any of them will be translated into an IDL interface. In general, we have chosen to model as interfaces the entities that the application will have to extend to interact with the Service. The remaining entities have been modelled as classes.

2.1.2.1 Infrastructure Module

The DCPS Infrastructure Module is comprised of the following classifiers:
- Entity
- DomainEntity
- QosPolicy
- Listener
- Status
- WaitSet
- Condition

![Class model of the DCPS Infrastructure Module](image)

Figure 2-5 Class model of the DCPS Infrastructure Module
• GuardCondition
• StatusCondition

2.1.2.1.1 Entity Class

This class is the abstract base class for all the DCPS objects that support QoS policies, a listener and a status condition.

<table>
<thead>
<tr>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>abstract set_qos</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>abstract get_qos</td>
</tr>
<tr>
<td>abstract set_listener</td>
</tr>
<tr>
<td>a_listener</td>
</tr>
<tr>
<td>mask</td>
</tr>
<tr>
<td>abstract get_listener</td>
</tr>
<tr>
<td>get_statuscondition</td>
</tr>
<tr>
<td>get_status_changes</td>
</tr>
<tr>
<td>enable</td>
</tr>
<tr>
<td>assert_liveliness</td>
</tr>
</tbody>
</table>

All operations except for set_qos, get_qos, set_listener, get_listener and enable may return the value NOT_ENABLED.

StatusKind is an enumerated type that identifies each concrete Status type.

The following sections explain all the operations in detail.

2.1.2.1.1.1 set_qos (abstract)

This operation is used to set the QoS policies of the Entity. This operation must be provided by each of the derived Entity classes (DomainParticipant, Topic, Publisher, DataWriter, Subscriber, DataReader) so that the policies that are meaningful to each Entity can be set.

The set of policies specified as the qos_list parameter are applied on top of the existing QoS, replacing the values of any policies previously set.

As described in Section 2.1.3, “Supported QoS,” on page 2-60, certain policies are “immutable”: they can only be set at Entity creation time, or before the entity is made enabled. If set_qos is invoked after the Entity is enabled and it attempts to change the value of an “immutable” policy, the operation will fail and it returns IMMUTABLE_POLICY.
Section 2.1.3, “Supported QoS,” on page 2-60 also describes that certain values of QoS policies can be incompatible with the settings of the other policies. The `set_qos` operation will also fail if it specifies a set of values that once combined with the existing values would result in an inconsistent set of policies. In this case, the return value is `INCONSISTENT_POLICY`.

The existing set of policies are only changed if the `set_qos` operation succeeds. This is indicated by the `OK` return value. In all other cases, none of the policies is modified.

Possible error codes returned in addition to the standard ones: `INCONSISTENT_POLICY`, `IMMUTABLE_POLICY`.

2.1.2.1.1.2 `get_qos` (abstract)

This operation allows access to the existing set of QoS policies for the `Entity`. This operation must be provided by each of the derived `Entity` classes (`DomainParticipant`, `Topic`, `Publisher`, `DataWriter`, `Subscriber`, `DataReader`) so that the policies meaningful to the particular `Entity` are retrieved.

2.1.2.1.1.3 `set_listener` (abstract)

This operation installs a `Listener` on the `Entity`. The listener will only be invoked on the changes of communication status indicated by the specified `mask`.

Only one listener can be attached to each `Entity`. If a listener was already set, the operation `set_listener` will replace it with the new one.

This operation must be provided by each of the derived `Entity` classes (`DomainParticipant`, `Topic`, `Publisher`, `DataWriter`, `Subscriber`, `DataReader`) so that the listener is of the concrete type suitable to the particular `Entity`.

2.1.2.1.1.4 `get_listener` (abstract)

This operation allows access to the existing `Listener` attached to the `Entity`.

This operation must be provided by each of the derived `Entity` classes (`DomainParticipant`, `Topic`, `Publisher`, `DataWriter`, `Subscriber`, `DataReader`) so that the listener is of the concrete type suitable to the particular `Entity`.

2.1.2.1.1.5 `get_statuscondition`

This operation allows access to the `StatusCondition` (Section 2.1.2.1.9) associated with the `Entity`. The returned condition can then be added to a `WaitSet` (Section 2.1.2.1.6) so that the application can wait for specific status changes that affect the `Entity`.

2.1.2.1.1.6 `get_status_changes`  
This operation retrieves the list of communication statuses in the `Entity` that are ‘triggered’. That is, the list of statuses whose value has changed since the last time the application read the status. The precise definition of the ‘triggered’ state of communication statuses is given in Section 2.1.4.2, “Changes in Status,” on page 2-79.
2.1.2.1.7 enable

This operation enables the Entity. All Entity objects are created disabled and must be enabled before the DCPS Service can use them.

Prior to enabling an Entity, the only operations that can be invoked on it are the ones to set or get the QoS policies and the listener and to get the StatusCondition. Other operations will return the error NOT_ENABLED.

2.1.2.1.8 assert_liveliness

This operation manually asserts the liveliness of the Entity. This is used in combination with the LIVELINESS QoS policy (cf. Section 2.1.3, “Supported QoS,” on page 2-60) to indicate to the Service that the entity remains active.

This operation needs only be used if the LIVELINESS setting is either MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC. Otherwise, it has no effect.

Note – Writing data via the write operation on a DataWriter asserts liveliness on the DataWriter itself, its Publisher and its DomainParticipant. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Complete details are provided in Section 2.1.3.7, “LIVELINESS,” on page 2-72.

2.1.2.1.2 DomainEntity Class

DomainEntity is the abstract base class for all DCPS entities, except for the DomainParticipant. Its sole purpose is to express that DomainParticipant is a special kind of Entity, which acts as a container of all other Entity, but itself cannot contain other DomainParticipant.

<table>
<thead>
<tr>
<th>DomainEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>no operations</td>
</tr>
</tbody>
</table>

2.1.2.1.3 QosPolicy Class

This class is the abstract root for all the QoS policies.

<table>
<thead>
<tr>
<th>QosPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| no operations |

It provides the basic mechanism for an application to specify quality of service parameters. It has an attribute name that is used to identify uniquely each QoS policy. All concrete QosPolicy classes derive from this root and include a value whose type depends on the concrete QoS policy.
The type of a \textit{QosPolicy} value may be atomic, such as an integer or float, or compound (a structure). Compound types are used whenever multiple parameters must be set coherently to define a consistent value for a \textit{QosPolicy}.

Each \textit{Entity} can be configured with a list of \textit{QosPolicy}. However, any \textit{Entity} cannot support any \textit{QosPolicy}. For instance, a \textit{DomainParticipant} supports different \textit{QosPolicy} than a \textit{Topic} or a \textit{Publisher}.

\textit{QosPolicy} can be set when the \textit{Entity} is created, or modified with the \textit{set_qos} method. Each \textit{QosPolicy} in the list is treated independently from the others. This approach has the advantage of being very extensible. However, there may be cases where several policies are in conflict. Consistency checking is performed each time the policies are modified via the \textit{set_qos} operation.

When a policy is changed after being set to a given value, it is not required that the new value be applied instantaneously; the Service is allowed to apply it after a transition phase. In addition, some \textit{QosPolicy} have “immutable” semantics meaning that they can only be specified either at \textit{Entity} creation time or else prior to calling the \textit{enable} operation on the \textit{Entity}.

Section 2.1.3, “Supported QoS,” on page 2-60 provides the list of all \textit{QosPolicy}, their meaning, characteristics and possible values, as well as the concrete \textit{Entity} to which they apply.

\subsection*{2.1.2.1.4 Listener Interface}
\textit{Listener} is the abstract root for all \textit{Listener} interfaces. All the supported kinds of concrete \textit{Listener} interfaces (one per concrete \textit{Entity}: \textit{DomainParticipant}, \textit{Topic}, \textit{Publisher}, \textit{DataWriter}, \textit{Subscriber}, and \textit{DataReader}) derive from this root and add methods whose prototype depends on the concrete \textit{Listener}.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Listener} \\
\hline
no attributes \\
no operations \\
\hline
\end{tabular}
\caption{Listener}
\end{table}

See Section 2.1.4.3, “Access through Listeners,” on page 2-81 for the list of defined listener interfaces. Listener interfaces provide a mechanism for the Service to asynchronously inform the application of relevant changes in the communication status.

\subsection*{2.1.2.1.5 Status Class}
\textit{Status} is the abstract root class for all communication status objects. All concrete kinds of \textit{Status} classes specialist this class.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Status} \\
\hline
no attributes \\
no operations \\
\hline
\end{tabular}
\caption{Status}
\end{table}
Each concrete Entity is associated with a set of Status objects whose value represents the “communication status” of that entity. These status values can be accessed with corresponding methods on the Entity. The changes on these status values are the ones that both cause activation of the corresponding StatusCondition objects and trigger invocation of the proper Listener objects to asynchronously inform the application.

Status objects and their relationship to Listener and Condition objects are detailed in Section 2.1.4.1, “Communication Status,” on page 2-76.

2.1.2.1.6 WaitSet Class

A WaitSet object allows an application to wait until one or more of the attached Condition objects has a trigger_value of TRUE or else until the timeout expires.

<table>
<thead>
<tr>
<th>WaitSet</th>
<th>no attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>attach_condition</td>
<td>operations</td>
</tr>
<tr>
<td>a_condition</td>
<td>ReturnCode_t</td>
</tr>
<tr>
<td>detach_condition</td>
<td>operations</td>
</tr>
<tr>
<td>a_condition</td>
<td>Condition</td>
</tr>
<tr>
<td>wait</td>
<td>operations</td>
</tr>
<tr>
<td>out: active_conditions</td>
<td>Condition []</td>
</tr>
<tr>
<td>timeout</td>
<td>operations</td>
</tr>
<tr>
<td>timeout</td>
<td>Duration_t</td>
</tr>
</tbody>
</table>

WaitSet 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). This is because it is not necessarily associated with a single DomainParticipant and could be used to wait on Condition objects associated with different DomainParticipant objects.

The following sections explain all the operations in detail.

2.1.2.1.6.1 attach_condition

Attaches a Condition to the WaitSet.

It is possible to attach a Condition on a WaitSet that is currently being waited upon (via the wait operation). In this case, if the Condition has a trigger_value of TRUE, then attaching the condition will unblock the WaitSet.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.1.6.2 detach_condition

Detaches a Condition from the WaitSet.

If the Condition was not attached to the WaitSet the operation will return BAD_PARAMETER.

Possible error codes returned in addition to the standard ones: BAD_PARAMETER.
2.1.2.1.6.3 wait

This operation allows an application thread to wait for the occurrence of certain conditions. If none of the conditions attached to the WaitSet have a trigger_value of TRUE, the wait operation will block suspending the calling thread.

The result of the wait operation is the list of all the attached conditions that have a trigger_value of TRUE (i.e., the conditions that unblocked the wait).

The wait operation takes a timeout argument that specifies the maximum duration for the wait. If this duration is exceeded and none of the attached Condition objects is true, wait will also return with the return code OK. In this case, the resulting list of conditions will be empty.

It is not allowed for more than one application thread to be waiting on the same WaitSet. If the wait operation is invoked on a WaitSet that already has a thread blocking on it, the operation will return immediately with the value PRECONDITION_NOT_MET.

2.1.2.1.7 Condition Class

A Condition is a root class for all the conditions that may be attached to a WaitSet. This basic class is specialized in three classes that are known by the middleware: GuardCondition (Section 2.1.2.1.8), StatusCondition (Section 2.1.2.1.9), and ReadCondition (Section 2.1.2.1.9).

<table>
<thead>
<tr>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>get_trigger_value</td>
</tr>
</tbody>
</table>

A Condition has a trigger_value that can be TRUE or FALSE and is set automatically by the Service.

2.1.2.1.7.1 get_trigger_value

This operation retrieves the trigger_value of the StatusCondition.

2.1.2.1.8 GuardCondition Class

A GuardCondition object is a specific Condition whose trigger_value is completely under the control of the application.

<table>
<thead>
<tr>
<th>GuardCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>enabled_statuses</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>set_trigger_value</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>
GuardCondition 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. When first created the trigger_value is set to FALSE.

The purpose of the GuardCondition is to provide the means for the application to manually wakeup a WaitSet. This is accomplished by attaching the GuardCondition to the WaitSet and then setting the trigger_value by means of the set_trigger_value operation.

2.1.2.1.8.1 set_trigger_value
This operation sets the trigger_value of the GuardCondition.

WaitSet objects behavior depends on the changes of the trigger_value of their attached conditions. Therefore, any WaitSet to which is attached the GuardCondition is potentially affected by this operation.

2.1.2.1.9 StatusCondition Class
A StatusCondition object is a specific Condition that is associated with each Entity.

<table>
<thead>
<tr>
<th>StatusCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>enabled_statuses</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>set_enabled_statuses</td>
</tr>
<tr>
<td>mask</td>
</tr>
</tbody>
</table>

The trigger_value of the StatusCondition depends on the communication status of that entity (e.g., arrival of data, loss of information, etc.), ‘filtered’ by the set of enabled_statuses on the StatusCondition.

The enabled_statuses and its relation to Listener and WaitSet is detailed in Trigger State of the StatusCondition.

2.1.2.1.9.1 set_enabled_statuses
This operation defines the list of communication statuses that are taken into account to determine the trigger_value of the StatusCondition. This operation may change the trigger_value of the StatusCondition.

WaitSet objects behavior depend on the changes of the trigger_value of their attached conditions. Therefore, any WaitSet to which the StatusCondition is attached is potentially affected by this operation.

If this function is not invoked, the default list of enabled statuses includes all the statuses.
2.1.2.2 Domain Module

The DCPS Domain Module is comprised of the following classes:

- DomainParticipant
- DomainParticipantFactory
- DomainParticipantListener

2.1.2.2.1 DomainParticipant Class

The DomainParticipant object plays several roles:

- It acts as a container for all other Entity objects.
- It acts as factory for the Publisher, Subscriber, Topic and MultiTopic Entity objects.
- It represents the participation of the application on a communication plane that isolates applications running on the same set of physical computers from each other. A domain establishes a “virtual network” linking all applications that share the same domainId and isolating them from applications running on different domains. In this way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.
It provides administration services in the domain, offering operations that allow the application to ‘ignore’ locally any information about a given participant (ignore_participant), publication (ignore_publication), subscription (ignore_subscription) or topic (ignore_topic).

<table>
<thead>
<tr>
<th>no attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
</tr>
<tr>
<td>create_publisher</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>delete_publisher</td>
</tr>
<tr>
<td>a_publisher</td>
</tr>
<tr>
<td>create_subscriber</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>delete_subscriber</td>
</tr>
<tr>
<td>a_subscriber</td>
</tr>
<tr>
<td>create_topic</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>type_name</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>delete_topic</td>
</tr>
<tr>
<td>a_topic</td>
</tr>
<tr>
<td>create_contentfilteredtopic</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>related_topic</td>
</tr>
<tr>
<td>filter_expression</td>
</tr>
<tr>
<td>expression_parameters</td>
</tr>
<tr>
<td>delete_contentfilteredtopic</td>
</tr>
<tr>
<td>a_contentfilteredtopic</td>
</tr>
<tr>
<td>create_multitopic</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>type_name</td>
</tr>
<tr>
<td>subscription_expression</td>
</tr>
<tr>
<td>expression_parameters</td>
</tr>
<tr>
<td>delete_multitopic</td>
</tr>
</tbody>
</table>

9. The actual format of the domainId is middleware specific. From the application point of view, it is a configuration parameter that appears only when the DomainParticipant is created.
The following sections explain all the operations in detail.

All the operations except the ones defined at the base-class level (namely, set_qos, get_qos, set_listener, get_listener and enable) may return the value NOT_ENABLED.

2.1.2.2.1.1 create_publisher
This operation creates a Publisher with the desired QoS policies and attaches to it the specified PublisherListener.

If the specified QoS policies are not compatible, the operation will fail and no Publisher will be created.

The created Publisher belongs to the DomainParticipant that is its factory.

2.1.2.2.1.2 delete_publisher
This operation deletes an existing Publisher.

A Publisher cannot be deleted if it has any attached DataWriter objects. If delete_publisher is called on a Publisher with existing DataWriter objects it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.1.3 create_subscriber
This operation creates a Subscriber with the desired QoS policies and attaches to it the specified SubscriberListener.

If the specified QoS policies are not compatible, the operation will fail and no Subscriber will be created.

The created Subscriber belongs to the DomainParticipant that is its factory.
2.1.2.2.1.4 delete_subscriber

This operation deletes an existing Subscriber.

A Subscriber cannot be deleted if it has any attached DataReader objects. If the delete_subscriber operation is called on a Subscriber with existing DataReader objects, it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.2.1.5 create_topic

This operation creates a Topic with the desired QoS policies and attaches to it the specified TopicListener.

If the specified QoS policies are not compatible, the operation will fail and no Topic will be created.

The created Topic belongs to the DomainParticipant that is its factory.

The Topic is bound to a type described by the type_name argument. Prior to creating a Topic the type must have been registered with the Service. This is done using the register_type operation on a derived class of the DataType interface as described in Section 2.1.2.3.6.

The application is not allowed to create two Topic objects with the same name attached to the same DomainParticipant. If the application attempts this, create_topic will fail and return an error.

2.1.2.2.1.6 delete_topic

This operation deletes a Topic.

The deletion of a Topic is not allowed if there are any existing DataReader, DataWriter, ContentFilteredTopic, or MultiTopic objects that are using the Topic. If the delete_topic operation is called on a Topic with any of these existing objects attached to it, it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.2.1.7 create_contentfilteredtopic

This operation creates a ContentFilteredTopic. As described in Section 2.1.2.3, “Topic-Definition Module,” on page 2-26, a ContentFilteredTopic can be used to do content-based subscriptions.

The related Topic being subscribed to is specified by means of the topic_name parameter. The ContentFilteredTopic only relates to samples published under that Topic, filtered according to their content. The filtering is done by means of evaluating a logical expression that involves the values of some of the data-fields in the sample. The logical expression derived from the filter_expression and expression_parameters arguments.

The syntax of the filter expression and parameters is described in Appendix A.
2.1.2.2.1.8 delete_contentfilteredtopic

This operation deletes a ContentFilteredTopic.

The deletion of a ContentFilteredTopic is not allowed if there are any existing DataReader objects that are using the ContentFilteredTopic. If the delete_contentfilteredtopic operation is called on a ContentFilteredTopic with existing DataReader objects attached to it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.2.1.9 create_multitopic

This operation creates a MultiTopic. As described in Section 2.1.2.3, “Topic-Definition Module,” on page 2-26 a MultiTopic can be used to subscribe to multiple topics and combine/filter the received data into a resulting type. In particular, MultiTopic provides a content-based subscription mechanism.

The resulting type is specified by the type_name argument. The list of topics and the logic used to combine filter and re-arrange the information from each Topic are specified using the subscription_expression and expression_parameters arguments.

The syntax of the expression and parameters is described in Appendix A.

2.1.2.2.1.10 delete_multitopic

This operation deletes a MultiTopic.

The deletion of a MultiTopic is not allowed if there are any existing DataReader objects that are using the MultiTopic. If the delete_multitopic operation is called on a MultiTopic with existing DataReader objects attached to it will return PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.2.1.11 lookup_topic

The operation lookup_topic gives access to an existing (or ready to exist) Topic, based on its name. The operation takes as arguments the name of the Topic and a timeout.

If a Topic of the same name already exists, it gives access to it, otherwise it waits (blocks the caller) until another mechanism creates it (or the specified timeout occurs). This other mechanism can be another thread, a configuration tool, or some other middleware service. Note that the Topic is a local object10 that acts as a 'proxy' to designate the global concept of topic. Middleware implementations could choose to propagate topics and make remotely created topics locally available.

If the operation times-out, a ‘nil’ value (as specified by the platform) is returned.

---

10. All the objects that make up this specification are local objects that are actually proxies to the service to be used by the application.
2.1.2.1.12 **get_builtin_subscriber**

This operation allows access to the built-in **Subscriber**. Each **DomainParticipant** contains several built-in **Topic** objects as well as corresponding **DataReader** objects to access them. All these **DataReader** objects belong to a single built-in **Subscriber**.

The built-in Topics are used to communicate information about other **DomainParticipant**, **Topic**, **DataReader**, and **DataWriter** objects. These built-in objects are described in Section 2.1.5, “Built-in Topics,” on page 2-88.

2.1.2.1.13 **ignore_participant**

This operation allows an application to instruct the Service to locally ignore a remote domain participant. From that point onwards the Service will locally behave as if the remote participant did not exist. This means it will ignore any **Topic**, publication, or subscription that originates on that domain participant.

This operation can be used, in conjunction with the discovery of remote participants offered by means of the “DCPSParticipant” built-in **Topic**, to provide e.g., access control. Application data can be associated with a **DomainParticipant** by means of the USER_DATA QoS policy. This application data is propagated as a filed in the built-in topic and can be used by an application to implement its own access control policy. See Section 2.1.5, “Built-in Topics,” on page 2-88 for more details on the built-in topics.

The domain participant to ignore is identified by the **handle** argument. This handle is the one that appears in the **SampleInfo** retrieved when reading the data-samples available for the built-in **DataReader** to the “DCPSParticipant” topic. The built-in **DataReader** is read with the same **read**/**take** operations used for any **DataReader**. These data-accessing operations are described in Section 2.1.2.5, “Subscription Module,” on page 2-43.

The **ignore_participant** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.1.14 **ignore_topic**

This operation allows an application to instruct the Service to locally ignore a **Topic**. This means it will locally ignore any, publication, or subscription to the **Topic**.

This operation can be used to save local resources when the application knows that it will never publish or subscribe to data under certain topics.

The **Topic** to ignore is identified by the **handle** argument. This handle is the one that appears in the **SampleInfo** retrieved when reading the data-samples from the built-in **DataReader** to the “DCPSTopic” topic.

The **ignore_topic** operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.
2.1.2.2.15 ignore_publication

This operation allows an application to instruct the Service to locally ignore a remote publication; a publication is defined by the association of a topic name, and user data and partition set on the Publisher (see the “DCPSPublication” built-in Topic in Section 2.1.5, “Built-in Topics,” on page 2-88). After this call, any data written related to that publication will be ignored.

The publication to ignore is identified by the handle argument. This handle is the one that appears in the SampleInfo retrieved when reading the data-samples from the built-in DataReader to the “DCPSPublication” topic.

The ignore_publication operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.16 ignore_subscription

This operation allows an application to instruct the Service to locally ignore a remote subscription; a subscription is defined by the association of a topic name, and user data and partition set on the Subscriber (see the “DCPSSubscription” built-in Topic in Section 2.1.5, “Built-in Topics,” on page 2-88). After this call, any data received related to that subscription will be ignored.

The DataReader to ignore is identified by the handle argument. This handle is the one that appears in the SampleInfo retrieved when reading the data-samples from the built-in DataReader to the “DCPSSubscription” topic.

The ignore_subscription operation is not required to be reversible. The Service offers no means to reverse it.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.2.2 DomainParticipantFactory Class

The sole purpose of this class is to allow the creation and destruction of DomainParticipant objects. DomainParticipantFactory itself has no factory. It is either a pre-existing object or it is created using some middleware specific API.

<table>
<thead>
<tr>
<th>DomainParticipantFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>create_participant</td>
</tr>
<tr>
<td>domainId</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>delete_participant</td>
</tr>
<tr>
<td>a_participant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_participant</td>
</tr>
<tr>
<td>domainId</td>
</tr>
<tr>
<td>qos_list</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>delete_participant</td>
</tr>
<tr>
<td>a_participant</td>
</tr>
</tbody>
</table>
The following sections give details about the operations.

2.1.2.2.2.1 create_participant

This operation creates a new DomainParticipant object. The DomainParticipant signifies that the calling application intends to join the Domain identified by the domainId argument.

2.1.2.2.2 delete_participant

This operation deletes an existing DomainParticipant. This operation can only be invoked if all domain entities belonging to the participant have already been deleted. Otherwise the error PRECONDITION_NOT_MET is returned.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.2.3 DomainParticipantListener Interface

This is the interface that can be implemented by an application-provided class and then registered with the DomainParticipant such that the application can be notified by the DCPS Service of relevant status changes.

The DomainParticipantListener interface extends all other Listener interfaces and has no additional operation beyond the ones defined by the more general listeners.

<table>
<thead>
<tr>
<th>DomainParticipantListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_inconsistent_topic_status</td>
</tr>
<tr>
<td>topic</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_liveliness_lost</td>
</tr>
<tr>
<td>the_writer</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_offered_deadline_missed</td>
</tr>
<tr>
<td>the_writer</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_offered_incompatible_qos</td>
</tr>
<tr>
<td>the_writer</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_data_on_readers</td>
</tr>
<tr>
<td>subscriber</td>
</tr>
<tr>
<td>on_sample_lost</td>
</tr>
<tr>
<td>subscriber</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_data_available</td>
</tr>
<tr>
<td>the_reader</td>
</tr>
<tr>
<td>on_sampleRejected</td>
</tr>
</tbody>
</table>
The purpose of the **DomainParticipantListener** is to be the listener of last resort that is notified of all status changes not captured by more specific listeners attached to the **DomainEntity** objects. When a relevant status change occurs, the DCPS Service will first attempt to notify the listener attached to the concerned **DomainEntity** if one is installed. Otherwise, the DCPS Service will notify the **Listener** attached to the **DomainParticipant**.

The relationship between listeners is described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

### 2.1.2.3 Topic-Definition Module

![Class model of the DCPS Topic-definition Module](image-url)

**Figure 2-7** Class model of the DCPS Topic-definition Module
The Topic-Definition Module is comprised of the following classes:

- TopicDescription
- Topic
- ContentFilteredTopic
- MultiTopic
- TopicListener
- DataType

2.1.2.3.1 TopicDescription Class
This class is an abstract class. It is the base class for Topic ContentFilteredTopic and MultiTopic.

<table>
<thead>
<tr>
<th>TopicDescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>type_name</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>no operations</td>
</tr>
</tbody>
</table>

TopicDescription represents the fact that both publications and subscriptions are tied to a single data-type. Its attribute type_name defines a unique resulting type for the publication or the subscription and therefore creates an implicit association with a DataType. TopicDescription has also a name that allows it to be retrieved locally.

2.1.2.3.2 Topic Class
Topic is the most basic description of the data to be published and subscribed.

<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>type_name (inherited)</td>
</tr>
<tr>
<td>name (inherited)</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>get_inconsistent_topic_status</td>
</tr>
</tbody>
</table>

A Topic is identified by its name, which must be unique in the whole Domain. In addition (by virtue of extending TopicDescription) it fully specifies the type of the data that can be communicated when publishing or subscribing to the Topic.

Topic is the only TopicDescription that can be used for publications and therefore associated to a DataWriter.

All operations except for the base-class operations set_qos, get_qos, set_listener, get_listener and enable may return the value NOT_ENABLED.
The following sections describe its operations.

2.1.2.3.2.1 get_inconsistent_topic_status

This method allows the application to retrieve the INCONSISTENT_TOPIC status of the Topic.

Each DomainEntity has a set of relevant communication statuses. A change of status causes the corresponding Listener to be invoked and can also be monitored by means of the associated StatusCondition.

The complete list of communication status, their values, and the DomainEntities they apply to is provided in Section 2.1.4.1, “Communication Status,” on page 2-76.

2.1.2.3.3 ContentFilteredTopic Class

ContentFilteredTopic is a specialization of TopicDescription that allows for content-based subscriptions.

<table>
<thead>
<tr>
<th>ContentFilteredTopic</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>type_name (inherited)</td>
</tr>
<tr>
<td>name (inherited)</td>
</tr>
<tr>
<td>related_topic</td>
</tr>
<tr>
<td>filter_expression</td>
</tr>
<tr>
<td>expression_parameters</td>
</tr>
</tbody>
</table>

No operations

ContentFilteredTopic describes a more sophisticated subscription that indicates the subscriber does not want to necessarily see all values of each instance published under the Topic. Rather, it wants to see only the values whose contents satisfy certain criteria. This class therefore can be used to request content-based subscriptions.

The selection of the content is done using the filter_expression with parameters filter_parameters.

- The filter_expression attribute is a string that specifies the criteria to select the data samples of interest. It is similar to the WHERE part of an SQL clause.

- The expression_parameters attribute is a sequence of strings that give values to the 'parameters' (i.e. "%n" tokens) in the filter_expression. The number of supplied parameters must fit with the requested values in the filter_expression (i.e., the number of %n tokens).

Appendix A describes the syntax of filter_expression and filter_parameters.
2.1.2.3.4 MultiTopic Class [optional]

MultiTopic is a specialization of TopicDescription that allows subscriptions that combine/filter/rearrange data coming from several topics.

| MultiTopic |
|---|---|
| attributes | |
| type_name (inherited) | string |
| name (inherited) | string |
| subscription_expression | string |
| expression_parameters | string [] |
| No operations | |

MultiTopic allows a more sophisticated subscription that can select and combine data received from multiple topics into a single resulting type (specified by the inherited type_name). The data will then be filtered (selection) and possibly re-arranged (aggregation/projection) according to a subscription_expression with parameters expression_parameters.

- The subscription_expression is a string that identifies the selection and re-arrangement of data from the associated topics. It is similar to a SQL clause where the SELECT part provides the fields to be kept, the FROM part provides the names of the topics that are searched for those fields\(^{11}\), and the WHERE clause gives the content filter. The Topics combined may have different types but they are restricted in that the type of the fields used for the NATURAL JOIN operation must be the same.

- The expression_parameters attribute is a sequence of strings that give values to the 'parameters' (i.e. "%n" tokens) in the subscription_expression. The number of supplied parameters must fit with the requested values in the filter_expression (i.e. the number of %n tokens).

- DataReader entities associated with a MultiTopic are alerted of data modifications by the usual Listener or Condition mechanisms (see Section 2.1.4) whenever modifications occur to the data associated with any of the topics relevant to the MultiTopic.

Appendix A describes the syntax of subscription_expression and expression_parameters.

---

\(^{11}\)It should be noted that in that case, the source for data may not be restricted to a single topic.
2.1.2.3.5 TopicListener Interface

Since Topic is a kind of Entity, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type TopicListener.

<table>
<thead>
<tr>
<th>TopicListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_inconsistent_topic_status</td>
</tr>
<tr>
<td>in topic Topic</td>
</tr>
<tr>
<td>in status InconsistentTopicStatus</td>
</tr>
</tbody>
</table>

2.1.2.3.6 DataType Interface

The DataType interface is an abstract interface that has to be specialized for each concrete type that will be used by the application.

It is required that each implementation of the Service provides an automatic means to generate this type-specific class from a description of the type (using IDL for example in the CORBA mapping). A DataType must be registered using the register_type operation on this type-specific class before it can be used to create Topic objects.

<table>
<thead>
<tr>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>register_type ReturnCode_t</td>
</tr>
<tr>
<td>in participant DomainParticipant</td>
</tr>
<tr>
<td>in type_name string</td>
</tr>
</tbody>
</table>

2.1.2.3.6.1 register_type

This operation allows an application to communicate to the Service the existence of a data type. The generated implementation of that operation embeds all the knowledge that has to be communicated to the middleware in order to make it able to manage the contents of data of that data type. This includes in particular the key definition that will allow the Service to distinguish different instances of the same type.

Possible error codes returned in addition to the standard ones: OUT_OF_RESOURCES.

2.1.2.3.7 Derived Classes for Each Application Class

For each data class defined by the application, there is a number of specialized classes that are required to facilitate the type-safe interaction of the application with the Service.

It is required that each implementation of the Service provides an automatic means to generate all these type-specific classes. DataType is one of the interfaces that these automatically-generated classes must implement. The complete set of automatic classes created for a hypothetical application named “Foo” are shown in Figure 2-8.
Figure 2-8  Classes auto-created for an application data type named Foo

This class must override the operation register_type and actually perform the registration of the "Foo" type with the service.

```
<<interface>>
DataType
  register_type()
```

```
FooDataType
  register_type()
  .
  .
  .
```

```
FooDataWriter
  register_instance()
  unregister_instance()
  dispose()
  dispose_w_timestamp()
  write()
  write_w_timestamp()
  get_key()
```

```
FooDataReader
  read()
  take()
  read_w_condition()
  take_w_condition()
  get_key()
```

```
Foo
```

```
<<comment>>
This class must override the operation register_type and actually perform the registration of the "Foo" type with the service.
```
2.1.2.4 Publication Module

The DCPS Publication Module is comprised of the following classifiers:

- Publisher
- DataWriter
- PublisherListener
- DataWriterListener

Figure 2-9 Class model of the DCPS Publication Module
2.1.2.4.1 Publisher Class

A Publisher is the object responsible for the actual dissemination of publications.

![Publisher Class Table]

The Publisher acts on the behalf of one or several DataWriter objects that belong to it. When it is informed of a change to the data associated with one of its DataWriter objects, it decides when it is appropriate to actually send the data-update message. In making this decision, it considers any extra information that goes with the data (timestamp, writer, etc.) as well as the QoS of the Publisher and the DataWriter.

All operations except for the base-class operations set_qos, get_qos, set_listener, get_listener and enable may return the value NOT_ENABLED.

2.1.2.4.1.1 set_listener (from Entity)

By virtue of extending Entity, a Publisher can be attached to a Listener at creation time or later by using the set_listener operation. The Listener attached must extend PublisherListener. Listeners are described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

2.1.2.4.1.2 get_listener (from Entity)

Retrieves the attached PublisherListener.

2.1.2.4.1.3 set_qos (from Entity)

By virtue of extending Entity, a Publisher can be given QoS at creation time or later by using the set_qos operation. Cf. Section 2.1.3, “Supported QoS,” on page 2-60 for the QoS policies that may be set on a Publisher.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.
2.1.2.4.1.4 get_qos (from Entity)
Allows access to the values of the QoS.

2.1.2.4.1.5 create_datawriter
This operation creates a DataWriter. The returned DataWriter will be attached and belongs to the Publisher.

The DataWriter returned by the create_datawriter operation will in fact be a derived class, specific to the data-type associated with the Topic. As described in Section 2.1.2.3.7, for each application-defined type “Foo” there is an implied, auto-generated class FooDataWriter that extends DataWriter and contains the operations to write data of type “Foo.”

2.1.2.4.1.6 delete_datawriter
This operation deletes a DataWriter that belongs to the Publisher. If the DataWriter does not belong to the Publisher, the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.4.1.7 lookup_datawriter
This operation retrieves a previously created DataWriter belonging to the Publisher that is attached to a Topic with a matching topic_name. If no such DataWriter exists, the operation will return ENTITY_NULL.

If multiple DataWriter attached to the Publisher satisfy this condition, then the operation will return one of them. It is not specified which one.

2.1.2.4.1.8 suspend_publications
This operation indicates to the Service that the application is about to make multiple modifications using DataWriter objects belonging to the Publisher.

It is a hint to the Service so it can optimize its performance by e.g., holding the dissemination of the modifications and then batching them.

It is not required that the Service uses this hint in any way.

The use of this operation must be matched by a corresponding call to resume_publications indicating that the set of modifications has completed.

2.1.2.4.1.9 resume_publications
This operation indicates to the Service that the application has completed the multiple changes initiated by the previous suspend_publications. This is a hint to the Service that can be used by a Service implementation to e.g., batch all the modifications made since the suspend_publications.

The call to resume_publications must match a previous call to suspend_publications. Otherwise the operation will return the error PRECONDITION_NOT_MET.
Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.4.1.10 begin_coherent_changes

This operation requests that the application will begin a ‘coherent set’ of modifications using DataWriter objects attached to the Publisher. The ‘coherent set’ will be completed by a matching call to end_coherent_changes.

A ‘coherent set’ it is a set of modifications that must be propagated in such a way that they are interpreted at the receivers’ side as a consistent set of modifications; that is, the receiver will only be able to access the data after all the modifications in the set are available at the receiver end\(^\text{12}\).

These calls can be nested. In that case, the coherent set terminates only with the last call to end_coherent_changes.

The support for ‘coherent changes’ enables a publishing application to change the value of several data-instances that could belong to the same or different topics and have those changes be seen ‘atomically’ by the readers. This is useful in cases where the values are inter-related (for example, if there are two data-instances representing the ‘altitude’ and ‘velocity vector’ of the same aircraft and both are changed, if may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

2.1.2.4.1.11 end_coherent_changes

This operation terminates the ‘coherent set’ initiated by the matching call to begin_coherent_changes. If there is no matching call to begin_coherent_changes the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET

\(^{12}\)This does not imply that the middleware has to encapsulate all the modifications in a single message; it only implies that the receiving applications will behave as if this was the case.
2.1.2.4.2 DataWriter Class

DataWriter allows the application to set the value of the data to be published under a given Topic.

<table>
<thead>
<tr>
<th>DataWriter</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>register_instance</td>
<td>InstanceHandle_t</td>
</tr>
<tr>
<td>unregister_instance</td>
<td>ReturnCode_t, Data</td>
</tr>
<tr>
<td>get_key_value</td>
<td>ReturnCode_t, Data</td>
</tr>
<tr>
<td>write</td>
<td>ReturnCode_t, Data</td>
</tr>
<tr>
<td>write_w_timestamp</td>
<td>ReturnCode_t, Data, Time_t</td>
</tr>
<tr>
<td>dispose</td>
<td>ReturnCode_t, Data</td>
</tr>
<tr>
<td>dispose_w_timestamp</td>
<td>ReturnCode_t, Data, Time_t</td>
</tr>
<tr>
<td>get_liveliness_lost_status</td>
<td>LivelinessLostStatus</td>
</tr>
<tr>
<td>get_offered_deadline_missed_status</td>
<td>OfferedDeadlineMissedStatus</td>
</tr>
<tr>
<td>get_offered_incompatible_qos_status</td>
<td>OfferedIncompatibleQosStatus</td>
</tr>
</tbody>
</table>

A DataWriter is attached to exactly one Publisher which acts as a factory for it.

A DataWriter is bound to exactly one Topic and therefore to exactly one data type. The Topic must exist prior to the DataWriter’s creation.

DataWriter is an abstract class. It must be specialized for each particular application data-type as shown in Figure 8. The additional methods that must be defined in the auto-generated class for a hypothetical application type “Foo” are shown in the table below:
All operations except for the base-class operations `set_qos`, `get_qos`, `set_listener`, `get_listener` and `enable` may return the value NOT_ENABLED.

The following sections provide details on the methods.

2.1.2.4.2.1 set_listener (from Entity)

By virtue of extending Entity, a DataWriter can be attached to a Listener at creation time or later by using the set_listener operation. The attached Listener must extend DataWriterListener. Listeners are described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

2.1.2.4.2.2 get_listener (from Entity)

Allows access to the attached DataWriterListener.
2.1.2.4.2.3 set_qos (from Entity)

By virtue of extending Entity, a DataWriter can be given QoS at creation time or later by using the set_qos operation. Cf. Section 2.1.3, “Supported QoS,” on page 2-60 for the QoS policies that may be set on a DataWriter.

The setting of QoS on the DataWriter results in a combination of the policies set at DataWriter level and of the ones set at the related Topic level. In case both Topic and DataWriter specify values for the same QosPolicy (identified by its name), the value specified by the DataWriter takes precedence. This applies after creation time as well; if the DataWriter does not specify a policy, the policy value will track changes in the Topic’s policy. To be more precise, for both Topic and DataWriter, the value of any QosPolicy can be either ‘set’ or ‘not set.’ The following table summarizes the resulting value of the policy depending on how it is specified for Topic and DataWriter:

<table>
<thead>
<tr>
<th>DataWriter setting</th>
<th>Topic setting</th>
<th>Resulting setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_SET</td>
<td>NOT_SET</td>
<td>Default value</td>
</tr>
<tr>
<td>NOT_SET</td>
<td>SET</td>
<td>Value set by Topic</td>
</tr>
<tr>
<td>SET</td>
<td>NOT_SET</td>
<td>Value set by DataWriter</td>
</tr>
<tr>
<td>SET</td>
<td>SET</td>
<td>Value set by DataWriter</td>
</tr>
</tbody>
</table>

In a sense, QoS set at Topic level represents a default setting that is inherited and can be ‘overridden’ by any DataWriter that refers to that Topic.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.4.2.4 get_qos (from Entity)

Allows access to the values of the QoS.

2.1.2.4.2.5 register_instance

This operation informs the Service that the application will be modifying a particular instance. It gives an opportunity to the Service to pre-configure itself to improve performance.

It takes as a parameter an instance (to get the key value) and returns a handle that can be used in successive write or dispose operations.

This operation should be invoked prior to calling any operation that modifies the instance, such as write, write_w_timestamp, dispose and dispose_w_timestamp.

The special value HANDLE_NIL may be returned by the Service if it does not want to allocate any handle for that instance.

The operation register_instance is idempotent. If it is called for an already registered instance, it just returns the already allocated handle. This may be used to lookup and retrieve the handle allocated to a given instance.
2.1.2.4.2.6 unregister_instance

This operation reverses the action of **register_instance**. It should only be called on an instance that is currently registered.

The operation **unregister_instance** should be called just one per instance, regardless of how many times **register_instance** was called for that instance.

This operation informs the Service that the DataWriter is not intending to modify any more of that data instance. This operation also indicates that the Service can locally remove all information regarding that instance. The application should not attempt to use the handle previously allocated to that instance after calling **unregister_instance**.

The special value HANDLE_NIL can be used for the parameter **handle**. This indicates that the identity of the instance should be automatically deduced from the **instance_data** (by means of the key).

If **handle** is any value other than HANDLE_NIL, then it must correspond to the value returned by **register_instance** when the instance (identified by its **key**) was registered. Then if there is no correspondence the result of the operation is unspecified.

If after that, the application wants to modify (write or dispose) the instance, it has to register it again, or else use the special **handle** value HANDLE_NIL.

This operation does not indicate that the instance is deleted (that is the purpose of **dispose**). The operation **unregister_instance** just indicates that the DataWriter no longer has ‘anything to say’ about the instance. DataReader objects that are reading the instance will eventually get an indication that their LIVENESS_CHANGED status (as defined in Section 2.1.4.1) has changed.

This operation can affect the ownership of the data instance (as described in Section 2.1.3.6). If the DataWriter was the exclusive owner of the instance, then calling **unregister_instance** will relinquish that ownership.

The operation must be called only on registered instances. Otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

2.1.2.4.2.7 get_key_value

This operation can be used to retrieve the instance key that corresponds to an **instance_handle**. The operation will only fill the fields that form the key inside the **key_holder** instance.

2.1.2.4.2.8 write

This operation modifies the value of a data instance. When this operation is used, the Service will automatically supply the value of the **source_timestamp** that is made available to DataReader objects by means of the **source_timestamp** attribute inside the SampleInfo (cf. Section 2.1.2.5, “Subscription Module,” on page 2-43 for more details on data timestamps at reader side and Section 2.1.3.11, “DESTINATION_ORDER,” on page 2-74 for the QoS policy DESTINATION_ORDER).
This operation must be provided on the specialized class that is generated for the particular application data-type that is being written. That way the data argument holding the data has the proper application-defined type (e.g., ‘Foo’).

As a side effect, this operation asserts liveliness on the DataWriter itself, the Publisher and the DomainParticipant.

The special value HANDLE_NIL can be used for the parameter handle. This indicates the identity of the instance should be automatically deduced from the instance_data (by means of the key).

If handle is any value other than HANDLE_NIL, then it must correspond to the value returned by register_instance when the instance (identified by its key) was registered. If there is no correspondence the result of the operation is unspecified.

2.1.2.4.2.9 write_w_timestamp

This operation performs the same function as write except that it also provides the value for the source_timestamp that is made available to DataReader objects by means of the source_timestamp attribute inside the SampleInfo (cf. Section 2.1.2.5, “Subscription Module,” on page 2-43 for more details on data timestamps at reader side and Section 2.1.3.11, “DESTINATION_ORDER,” on page 2-74 for the QoS policy DESTINATION_ORDER).

Similar to write, this operation must also be provided on the specialized class that is generated for the particular application data-type that is being written.

2.1.2.4.2.10 dispose

This operation requests the middleware to delete the data (the actual deletion is postponed until there is no more use for that data in the whole system). In general, applications are made aware of the deletion by means of operations on the DataReader objects that already knew that instance (cf. Section 2.1.2.5, “Subscription Module,” on page 2-43 for more details).

This operation does not modify the value of the instance. The instance parameter is passed just for the purposes of identifying the instance.

When this operation is used, the Service will automatically supply the value of the source_timestamp that is made available to DataReader objects by means of the source_timestamp attribute inside the SampleInfo.

In addition, in case the DURABILITY QoS policy is TRANSIENT or PERSISTENT, the Service should take care to clean anything related to that instance so that late-joining applications would not see it.

The special value HANDLE_NIL can be used for the parameter handle. This indicates the identity of the instance should be automatically deduced from the instance_data (by means of the key).

---

13.DataReader objects that didn't know the instance will never see it.
If **handle** is any value other than HANDLE_NIL, then it must correspond to the value returned by **register_instance** when the instance (identified by its **key**) was registered. Then if there is no correspondence the result of the operation is unspecified.

The operation must be only called on registered instances. Otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

### 2.1.2.4.2.11 dispose_w_timestamp

This operation performs the same functions as **dispose** except that the application provides the value for the **source_timestamp** that is made available to **DataReader** objects by means of the **source_timestamp** attribute inside the **SampleInfo** (cf. Section 2.1.2.5, “Subscription Module,” on page 2-43).

The operation must be only called on registered instances. Otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

### 2.1.2.4.2.12 get_liveliness_lost_status

This operation allows access to the LIVELINESS_LOST communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.

### 2.1.2.4.2.13 get_offered_deadline_missed_status

This operation allows access to the OFFERED_DEADLINE_MISSED communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.

### 2.1.2.4.2.14 get_offered_incompatible_qos_status

This operation allows access to the OFFERED_INCOMPATIBLE_QOS communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.
2.1.2.4.3 PublisherListener Interface

<table>
<thead>
<tr>
<th>PublisherListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>no operations</td>
</tr>
</tbody>
</table>

Since a Publisher is a kind of Entity, it has the ability to have a listener associated with it. In this case, the associated listener should be of concrete type PublisherListener. The use of this listener and its relationship to changes in the communication status of the Publisher is described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

2.1.2.4.4 DataWriterListener Interface

<table>
<thead>
<tr>
<th>DataWriterListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_liveliness_lost</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>on_offered_deadline_missed</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>on_offered_incompatible_qos</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Since a DataWriter is a kind of Entity, it has the ability to have a listener associated with it. In this case, the associated listener should be of concrete type DataWriterListener. The use of this listener and its relationship to changes in the communication status of the DataWriter is described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

2.1.2.4.5 Concurrency Behavior

This specification makes no assumption about the way the publishing application is designed. In particular, several DataWriter may operate in different threads. If they share the same Publisher, the middleware guarantees that its operations are thread-safe. However, it is not required that each requesting thread be treated in isolation from the others (leading e.g., to several isolated sets of coherent changes). If this is the desired behavior, the proper design is to create a Publisher for each thread.
2.1.2.5 Subscription Module

The Subscription Module is comprised of the following classifiers:

- Subscriber
- DataReader
- DataSample
- SampleInfo
- SubscriberListener
- DataReaderListener
- ReadCondition
- QueryCondition

Figure 2-10  Class model of the DCPS Subscription Module
The following section presents how the data can be accessed and introduces the lifecycle state. The 2.1.2.5.2 Subscriber Class section through to the QueryCondition Class section 2.1.2.5.9 provide details on each class belonging to that module.

2.1.2.5.1 Access to the data

Data is made available to the application by the following operations on DataReader objects: read, read_w_condition, take, and take_w_condition. The general semantics of the “read” operations is that the application only gets access to the corresponding data; the data remains the middleware’s responsibility and can be read again. The semantics of the “take” operations is that the application takes full responsibility for the data; that data will no longer be available locally to the middleware. Consequently, it is possible to access the same information multiple times only if all previous accesses were read operations, not take.

Each of these operations returns a collection of Data values and associated SampleInfo objects. Each data value represents an atom of data information (i.e., a value for one instance). The SampleInfo contains information pertaining to the associated Data value:

- The lifecycle state of the related instance (i.e., if the instance is NEW, MODIFIED, DISPOSED, or NO_WRITERS – see below).
- The sample state of the Data value (i.e., if it has already been read or not).
- The source (production) timestamp.

The lifecycle state of each sample indicates whether the application has already accessed—by means of the DataReader—a different (previous) sample for that instance (identified by the key). For a given instance, the valid sequences of lifecycle states in the samples of that instance returned to an application (by means of read or take) are illustrated in Figure 2-11. The “instance disposed by writer” represents the arrival to the Service of information regarding the fact that dispose_instance was called on the DataWriter that “owns” the instance, or by any DataWriter if OWNERSHIP is SHARED.

14.Meaning a precise instance value.
15.The concept of “ownership” is described in Section 2.1.3.6, “OWNERSHIP,” on page 2-71.
Figure 2-11  Legal sequences of lifecycle states for samples of a single instance

Note that this state machine does not define the state of a data instance—that is an implementation issue and is not specified. It simply defines the valid sequences of observable values of the lifecycle_state returned along with the samples accessed by the application via read or take. The observed value is represented in Figure 2-11 by means of the “action”\(^{16}\) associated with the read/take events.

The value of the lifecycle_state returned along with each sample have the following meaning:

- NEW. Indicates that this is the first sample ever read or taken by the application for that particular instance. Alternatively, this is the first sample read or taken for the instance since it was disposed.
- MODIFIED. Indicates that the application has read or taken other (different) samples of this instance prior to the one currently being examined.
- DISPOSED. Indicates that the instance was disposed by a writer. In the case where OWNERSHIP QoS (see Section 2.1.3.6, “OWNERSHIP,” on page 2-71) is EXCLUSIVE, this indicates the DataWriter that owns the instance has called dispose. In the case where OWNERSHIP is SHARED this indicates that one of the DataWriter objects has called dispose.

\(^{16}\)The “action” is the part that follows “/” on the expression attached to the transition. For example, the action on the “read/NO_WRITERS” expression at the bottom right-hand of the figure is “NO_WRITERS.”
• NO_WRITERS. Indicates that there are no longer any “live” (according to the LIVELINESS QoS) DataWriter objects writing the instance. To be more precise this means that the “live” DataWriter objects either have never written a value for the instance, or else they have called unregister_instance after the last time they wrote the instance.

The transition from DISPOSED to MODIFIED indicates that it is possible for an application to not detect the disposal of an instance. This only occurs in the case where HISTORY QoS kind is KEEP_LAST and then only if samples exceeding the HISTORY depth arrive after the instance is disposed before the application accesses the sample. This is reasonable since the KEEP_LAST indicates the application is accepting to miss intermediate modifications and the disposal was followed by samples indicating a subsequent creation with the same key value.

The transition from NO_WRITERS to MODIFIED models the case where the Service detects an instance that has no DataWriter objects but data arrives before the application accesses the instance. In this case the application will not see any samples with the NO_WRITERS lifecycle state. This is desirable since the NO_WRITERS was a temporary situation reversed by the arrival of data.

The application accesses data by means of the operations read or take on the DataReader. These operations return a collection of DataSamples consisting of a SampleInfo part and a Data part. The way the middleware builds this collection depends on QoS policies set on the DataReader and Subscriber, as well as the source timestamp of the samples, and the parameters passed to the read/take operations, namely:

• the desired lifecycle states (i.e., NEW, MODIFIED, DISPOSED, NO_WRITERS) or a combination of these);

• the desired sample states (i.e., READ, NOT_READ or both);

The read and take operations are non-blocking and just deliver what is currently available. However, the samples returned for each instance are subject to meeting the legal sequences of lifecycle states implied by Figure 2-11. This only affects the case where the HISTORY QoS policy (cf. Section 2.1.3.12, “HISTORY,” on page 2-75) is either KEEP_ALL, or has a history deeper than one sample.

The constraints imposed by the ‘legal’ lifecycle sequences mean that, for each instance, the middleware is not allowed to deliver MODIFIED or DISPOSED samples if it has not already delivered the corresponding NEW samples. Similarly it is not allowed to deliver DISPOSED samples if it has not delivered the corresponding NEW and MODIFIED samples ahead of it. This means that if the application asks to read all MODIFIED samples in a DataReader and an instance in the DataReader has samples with lifecycle state NEW ahead of those with lifecycle MODIFIED, then none of the samples in that instance will be part of the list returned by read. The same applies to take. The application must take the NEW samples before it can take the MODIFIED or DISPOSED ones.

---

17. i.e., the data-samples that are parts of the list as well as their order
The `read_w_condition` and `take_w_condition` operations take a `ReadCondition` object as a parameter instead of lifecycle or sample states. The behavior is that the samples returned will only be those for which the condition is TRUE. These operations, in conjunction with `ReadCondition` objects and a `WaitSet`, allow performing waiting reads (see below).

Once the data samples are available to the data readers, they can be read or taken by the application. The basic rule is that the application may do this in any order it wishes. This approach is very flexible and allows the application ultimate control. However, the application must use a specific access pattern in case it needs to retrieve samples in the proper order received, or it wants to access a complete set of coherent changes.

To access data coherently, or in order, the PRESENTATION QoS (explained in Section 2.1.3.3, “PRESENTATION,” on page 2-69) must be set properly and the application must conform to the access pattern described below. Otherwise, the application will still access the data but will not necessarily see all coherent changes together, nor will it see the changes in the proper order.

There is a general pattern that will provide both ordered and coherent access across multiple `DataReader`. This pattern will work for any settings of the PRESENTATION QoS. Simpler patterns may also work for specific settings of the QoS as described below.

1. **General pattern to access samples as a coherent set and/or in order across `DataReader` entities.** This case applies when PRESENTATION QoS specifies "`access_scope`=GROUP".
   - Upon notification to the `SubscriberListener` or following the similar `StatusCondition` enabled, the application uses `begin_access` on the `Subscriber` to indicate it will be accessing data through the `Subscriber`.
   - Then it calls `get_datareaders` on the `Subscriber` to get the list of `DataReader` objects where data samples are available.
   - Following this it calls `read` or `take` on each `DataReader` in the same order returned to access all the relevant changes in the `DataReader`.
   - Once it has called `read` or `take` on all the readers, it calls `end_access`.

   Note that if the PRESENTATION QoS policy specifies `ordered_access`=TRUE then the list of `DataReader` may return the same reader several times. In this manner the correct sample order can be maintained among samples in different `DataReader` objects.

2. **Specialized pattern if no order or coherence needs to be maintained across `DataReader` entities.** This case applies if PRESENTATION QoS policy specifies `access_scope` something other than GROUP.
   - In this case, it is not required for the application to call `begin_access` and `end_access`. However, doing so is not an error and it will have no effect.

18.I.e.: on `Subscriber` with mask referring to DATA_ON_READERS.
• The application accesses the data by calling `read` or `take`\(^{19}\) on each `DataReader` in any order it wishes.

• The application can still call `get_datareaders` to determine which readers have data to be read, but it does not need to read all of them, nor read them in a particular order. Furthermore, the return of `get_datareaders` will be logically a "set" in the sense that the same reader will not appear twice, and the order of the readers returned is not specified.

3. Specialized pattern if the application accesses the data within the `SubscriberListener`. This case applies regardless of the PRESENTATION QoS policy when the application accesses the data inside the listener’s implementation of the `on_data_on_readers` operation.

• Similar to the previous case (2 above), it is not required for the application to call `begin_access` and `end_access`, but doing so has no effect.

• The application can access data by calling `read` or `take`\(^{20}\) on each `DataReader` in any order it wishes.

• The application can also delegate the accessing of the data to the `DataReaderListener` objects installed on each `DataReader` by calling `notify_datareaders`.

• Similar to the previous case (2 above), the application can still call `get_datareaders` to determine which readers have data to be read, but it does not need to read all of them, nor read them in a particular order. Furthermore, the return of `get_datareaders` will be logically a 'set'.

2.1.2.5.2 Subscriber Class

A `Subscriber` is the object responsible for the actual reception of the data resulting from its subscriptions.

<table>
<thead>
<tr>
<th><code>Subscriber</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td><code>create_datareader</code></td>
</tr>
<tr>
<td><code>a_topic</code></td>
</tr>
<tr>
<td><code>qos</code></td>
</tr>
<tr>
<td><code>listener</code></td>
</tr>
<tr>
<td><code>delete_datareader</code></td>
</tr>
<tr>
<td><code>a_datareader</code></td>
</tr>
<tr>
<td><code>lookup_datareader</code></td>
</tr>
<tr>
<td><code>topic_name</code></td>
</tr>
</tbody>
</table>

19. Or the variants `read_w_condition` and `take_w_condition`.

20. Or the variants `read_w_condition` and `take_w_condition`. 
A **Subscriber** acts on the behalf of one or several **DataReader** objects that are related to it. When it receives data (from the other parts of the system), it builds the list of concerned **DataReader** objects, and then indicates to the application that data is available, through its listener or by enabling related conditions. The application can access the list of concerned **DataReader** objects through the operation **get_datareaders** and then access the data available though operations on the **DataReader**.

All operations except for the base-class operations **set_qos**, **get_qos**, **set_listener**, **get_listener** and **enable** may return the value NOT_ENABLED.

### 2.1.2.5.2.1 set_listener (from Entity)

By virtue of extending **Entity**, a **Subscriber** can be attached to a **Listener** at creation time or later by using the **set_listener** operation. The **Listener** attached must extend **SubscriberListener**. Listeners are described in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

### 2.1.2.5.2.2 get_listener (from Entity)

Allows access to the attached **SubscriberListener**.

### 2.1.2.5.2.3 set_qos (from Entity)

By virtue of extending **Entity**, a **Subscriber** can be given QoS at creation time or later by using the **set_qos** operation. Cf. Section 2.1.3, “Supported QoS,” on page 2-60 for the list of QoS policies that may be set on a **Subscriber**.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

### 2.1.2.5.2.4 get_qos (from Entity)

Allows access to the values of the QoS.

### 2.1.2.5.2.5 create_datareader

This operation creates a **DataReader**. The returned **DataReader** will be attached and belong to the **Subscriber**.
2

The DataReader returned by the create_datareader operation will in fact be a derived class, specific to the data-type associated with the Topic. As described in Section 2.1.2.3.7, for each application-defined type “Foo” there is an implied auto-generated class FooDataReader that extends DataReader and contains the operations to read data of type “Foo.”

2.1.2.5.2.6 delete_datareader
This operation deletes a DataReader that belongs to the Subscriber. If the DataReader does not belong to the Subscriber the operation returns the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET

2.1.2.5.2.7 lookup_datareader
This operation retrieves a previously-created DataReader belonging to the Subscriber that is attached to a Topic with a matching topic_name. If no such DataReader exists, the operation will return ENTITY_NULL.

If multiple DataReader attached to the Subscriber satisfy this condition, then the operation will return one of them. It is not specified which one.

The use of this operation on the built-in Subscriber allows access to the built-in DataReader entities for the built-in topics.

2.1.2.5.2.8 begin_access
This operation indicates that the application is about to access the data samples in any of the DataReader objects attached to the Subscriber.

The application is required to use this operation only if PRESENTATION QosPolicy of the Subscriber to which the DataReader belongs has the access_scope set to ‘GROUP’.

In the aforementioned case, the operation begin_access must be called prior to calling any of the sample-accessing operations, namely: get_datareaders on the Subscriber and read, take, read_w_condition, take_w_condition on any DataWriter. Otherwise the sample-accessing operations will return the error PRECONDITION_NOT_MET. Once the application has finished accessing the data samples it must call end_access.

It is not required for the application to call begin_access/end_access if the PRESENTATION QosPolicy has the access_scope set to something other than ‘GROUP’. Calling begin_access/end_access in this case is not considered an error and has no effect.

The calls to begin_access/end_access may be nested. In that case, the application must call end_access as many times as it called begin_access.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET.

21.Cf. Section 2.1.5 for more details on built-in topics.
2.1.2.5.2.9 end_access
Indicates that the application has finished accessing the data samples in DataReader objects managed by the Subscriber.

This operation must be used to ‘close’ a corresponding begin_access.

After calling end_access the application should no longer access any of the Data or SampleInfo elements returned from the sample-accessing operations. This call must close a previous call to begin_access otherwise the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones: PRECONDITION_NOT_MET.

2.1.2.5.2.10 get_datareaders
This operation allows the application to access the DataReader objects that contain samples with the specified lifecycle_states and sample_states.

If the PRESENTATION QosPolicy of the Subscriber to which the DataReader belongs has the access_scope set to ‘GROUP’. This operation should only be invoked inside a begin_access/end_access block. Otherwise it will return the error PRECONDITION_NOT_MET.

Depending on the setting of the PRESENTATION QoS policy (cf. Section 2.1.3.3, “PRESENTATION,” on page 2-69), the returned collection of DataReader objects may be a ‘set’ containing each DataReader at most once in no specified order, or a ‘list’ containing each DataReader one or more times in a specific order.

1. If PRESENTATION access_scope is INSTANCE or TOPIC the returned collection is a ‘set’.
2. If PRESENTATION access_scope is GROUP and ordered_access is set to TRUE, then the returned collection is a ‘list’. This difference is due to the fact that, in the second situation it is required to access samples belonging to different DataReader objects in a particular order. In this case, the application should process each DataReader in the same order it appears in the ‘list’ and read or take exactly one sample from each DataReader. The patterns that an application should use to access data is fully described in “Access to the data.”

2.1.2.5.2.11 notify_datareaders
This operation invokes the operation on_data_available on the DataReaderListener objects attached to contained DataReader entities containing samples with any LifecycleState and SampleState ‘NOT_READ’.

This operation is typically invoked from the on_data_on_readers operation in the SubscriberListener. That way the SubscriberListener can delegate to the DataReaderListener objects the handling of the data.
2.1.2.5.2.12 get_sample_lost_status

This operation allows access to the SAMPLE_LOST communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.

2.1.2.5.3 DataReader Class

A DataReader allows the application (1) to declare the data it wishes to receive (i.e., make a subscription) and (2) to access the data received by the attached Subscriber.

<table>
<thead>
<tr>
<th>DataReader</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>read</td>
</tr>
<tr>
<td>out: data_values</td>
</tr>
<tr>
<td>out: sample_infos</td>
</tr>
<tr>
<td>sample_states</td>
</tr>
<tr>
<td>lifecycle_states</td>
</tr>
<tr>
<td>take</td>
</tr>
<tr>
<td>out: data_values</td>
</tr>
<tr>
<td>out: sample_info</td>
</tr>
<tr>
<td>sample_states</td>
</tr>
<tr>
<td>lifecycle_states</td>
</tr>
<tr>
<td>read_w_condition</td>
</tr>
<tr>
<td>out: data_values</td>
</tr>
<tr>
<td>out: sample_info</td>
</tr>
<tr>
<td>a_condition</td>
</tr>
<tr>
<td>take_w_condition</td>
</tr>
<tr>
<td>out: data_values</td>
</tr>
<tr>
<td>out: sample_info</td>
</tr>
<tr>
<td>a_condition</td>
</tr>
<tr>
<td>get_key_value</td>
</tr>
<tr>
<td>inout: key_holder</td>
</tr>
<tr>
<td>handle</td>
</tr>
<tr>
<td>create_readcondition</td>
</tr>
<tr>
<td>sample_states</td>
</tr>
<tr>
<td>lifecycle_states</td>
</tr>
<tr>
<td>create_querycondition</td>
</tr>
<tr>
<td>sample_states</td>
</tr>
</tbody>
</table>
A **DataReader** refers to exactly one **TopicDescription** (either a **Topic**, a **ContentFilteredTopic** or a **MultiTopic**) that identifies the data to be read. The subscription has a unique resulting type. The data-reader may give access to several instances of the resulting type, which can be distinguished from each other by their *key* (as described in Section 2.1.1.2.2).

**DataReader** is an abstract class. It must be specialized for each particular application data-type as shown in Figure 8. The additional methods that must be defined in the auto-generated class for a hypothetical application type “Foo” are shown in the table below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>lifecycle_states</td>
<td>LifecycleStateKind []</td>
<td>read</td>
</tr>
<tr>
<td>query_expression</td>
<td>string</td>
<td>out: data_values Foo []</td>
</tr>
<tr>
<td>query_parameters</td>
<td>string []</td>
<td>out: sample_infos SampleInfo []</td>
</tr>
<tr>
<td>delete_readcondition</td>
<td>ReturnCode_t</td>
<td>lifecycle_states LifecycleStateKind []</td>
</tr>
<tr>
<td>a_condition</td>
<td>ReadCondition</td>
<td>sample_states SampleStateKind []</td>
</tr>
<tr>
<td>get_liveliness_changed_status</td>
<td>LivelinessChangedStatus</td>
<td></td>
</tr>
<tr>
<td>get_requested_deadline_missed_status</td>
<td>RequestedDeadlineMissedStatus</td>
<td></td>
</tr>
<tr>
<td>get_requested_incompatible_qos_status</td>
<td>RequestedIncompatibleQosStatus</td>
<td></td>
</tr>
</tbody>
</table>

**FooDataReader**

**operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td></td>
<td>read</td>
</tr>
<tr>
<td>out: data_values</td>
<td>Foo []</td>
<td>out: data_values Foo []</td>
</tr>
<tr>
<td>out: sample_infos</td>
<td>SampleInfo []</td>
<td>out: sample_infos SampleInfo []</td>
</tr>
<tr>
<td>lifecycle_states</td>
<td>LifecycleStateKind []</td>
<td>lifecycle_states LifecycleStateKind []</td>
</tr>
<tr>
<td>sample_states</td>
<td>SampleStateKind []</td>
<td>sample_states SampleStateKind []</td>
</tr>
<tr>
<td>take</td>
<td></td>
<td>take</td>
</tr>
<tr>
<td>out: data_values</td>
<td>Foo []</td>
<td>out: data_values Foo []</td>
</tr>
<tr>
<td>out: sample_info</td>
<td>SampleInfo []</td>
<td>out: sample_info SampleInfo []</td>
</tr>
<tr>
<td>lifecycle_states</td>
<td>LifecycleStateKind []</td>
<td>lifecycle_states LifecycleStateKind []</td>
</tr>
<tr>
<td>sample_states</td>
<td>SampleStateKind []</td>
<td>sample_states SampleStateKind []</td>
</tr>
<tr>
<td>read_w_condition</td>
<td></td>
<td>read_w_condition</td>
</tr>
<tr>
<td>out: data_values</td>
<td>Foo []</td>
<td>out: data_values Foo []</td>
</tr>
<tr>
<td>out: sample_info</td>
<td>SampleInfo []</td>
<td>out: sample_info SampleInfo []</td>
</tr>
<tr>
<td>a_condition</td>
<td>ReadCondition</td>
<td>a_condition ReadCondition</td>
</tr>
<tr>
<td>take_w_condition</td>
<td></td>
<td>take_w_condition</td>
</tr>
<tr>
<td>out: data_values</td>
<td>Foo []</td>
<td>out: data_values Foo []</td>
</tr>
<tr>
<td>out: sample_info</td>
<td>SampleInfo []</td>
<td>out: sample_info SampleInfo []</td>
</tr>
<tr>
<td>a_condition</td>
<td>ReadCondition</td>
<td>a_condition ReadCondition</td>
</tr>
<tr>
<td>get_key_value</td>
<td></td>
<td>get_key_value</td>
</tr>
<tr>
<td>inout: key_holder</td>
<td>Foo</td>
<td>inout: key_holder Foo</td>
</tr>
<tr>
<td>handle</td>
<td>InstanceHandle_t</td>
<td>handle InstanceHandle_t</td>
</tr>
</tbody>
</table>
All operations except for the base-class operations set_qos, get_qos, set_listener, get_listener and enable may return the error NOT_ENABLED.

All sample-accessing operations, namely: read, take, read_w_condition, take_w_condition may return the error PRECONDITION_NOT_MET. The circumstances that result on this are described in Section 2.1.2.5.2.8.

The following sections give details on all the operations.

2.1.2.5.3.1 set_listener (from Entity)
By virtue of extending Entity, a DataReader can be attached to a Listener at creation time or later by using the set_listener operation. The Listener attached must extend DataReaderListener. Listeners are described in Section 2.1.4.

2.1.2.5.3.2 get_listener (from Entity)
Allows access to the attached DataReaderListener.

2.1.2.5.3.3 set_qos (from Entity)
By virtue of extending Entity, a DataReader can be given QoS at creation time or later by using the set_qos operation. Cf. Section 2.1.3 for the list of QoS policies that may set on a DataReader.

The setting of QoS on the DataReader results in a combination of the policies set at DataReader level and of the ones set at the related Topic level. The algorithm used to resolve the case where values for the same QosPolicy are set both on the DataReader and the Topic is the same described for the DataWriter in Section 2.1.2.4.2.3.

Possible error codes returned in addition to the standard ones: IMMUTABLE_POLICY, INCONSISTENT_POLICY.

2.1.2.5.3.4 get_qos (from Entity)
Allows access to the values of the QoS

2.1.2.5.3.5 create_readcondition
This operation creates a ReadCondition. The returned ReadCondition will be attached and belong to the DataReader.

2.1.2.5.3.6 create_querycondition
This operation creates a QueryCondition. The returned QueryCondition will be attached and belong to the DataReader.

The syntax of the query_expression and query_parameters parameters is described in Appendix A.
2.1.2.5.3.7 delete_readcondition

This operation deletes a ReadCondition attached to the DataReader. Since QueryCondition specializes ReadCondition it can also be used to delete a QueryCondition. If the ReadCondition is not attached to the DataReader the operation will return the error PRECONDITION_NOT_MET.

Possible error codes returned in addition to the standard ones:
PRECONDITION_NOT_MET

2.1.2.5.3.8 read

This operation accesses a collection of data samples from the DataReader. Depending on the setting of the PRESENTATION QoS policy (cf. Section 2.1.3.3), the operation will return either a ‘list’ of samples or else a single sample.

1. If PRESENTATION access_scope is INSTANCE, the returned collection is a ‘list’ where samples belonging to the same data-instance are consecutive.

2. If PRESENTATION access_scope is TOPIC and ordered_access is set to FALSE, then returned collection is a ‘list’ where samples belonging to the same data-instance are consecutive.

3. If PRESENTATION access_scope is TOPIC and ordered_access is set to TRUE, then the returned collection is a ‘list’ were samples belonging to the same instance may or may not be consecutive. This is because to preserve order it may be necessary to mix samples from different instances.

4. If PRESENTATION access_scope is GROUP and ordered_access is set to FALSE, then returned collection is a ‘list’ where samples belonging to the same data instance are consecutive.

5. If PRESENTATION access_scope is GROUP and ordered_access is set to TRUE, then the returned collection contains at most one sample. The difference in this case is due to the fact that is required that the application is able to read samples belonging to different DataReader objects in a specific order.

In any case, the relative order between the samples of one instance is consistent with the DESTINATION_ORDER QosPolicy.

The act of reading a sample changes its sample_state to READ but it does not affect the lifecycle_state of the instance.

This operation must be provided on the specialized class that is generated for the particular application data-type that is being read.

2.1.2.5.3.9 take

This operation accesses a collection of data-samples from the DataReader. The operation will return either a ‘list’ of samples or else a single sample. This is controlled by the PRESENTATION QosPolicy using the same logic as for the read operation (cf. Section 2.1.2.5.3.8)
The act of reading a sample removes it from the middleware so it cannot be ‘read’ or ‘taken’ again. It also may change the lifecycle_state of the instance.

Similar to read, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

2.1.2.5.3.10 read_w_condition
This operation accesses via ‘read’ the samples that match the criteria specified in the ReadCondition. This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

In case the ReadCondition is a ‘plain’ ReadCondition and not the specialized QueryCondition, the operation is equivalent to calling read and passing as lifecycle_states and sample_states the value of the corresponding attributes in the read_condition. Using this operation the application can avoid repeating the same parameters specified when creating the ReadCondition.

The samples are accessed via read and therefore this operation does not change the lifecycle_state of any instance and leaves the samples under the control of the Service so they can be accessed again.

Similar to read, this operation must be provided on the specialized class that is generated for the particular application data-type that is being read.

2.1.2.5.3.11 take_w_condition
This operation is the analogous to read_w_condition except it accesses samples via the ‘take’ operation.

This operation removes samples from the middleware so they cannot be ‘read’ or ‘taken’ again. It also may change the lifecycle_state of the instances whose samples are taken.

This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

Similar to take, this operation must be provided on the specialized class that is generated for the particular application data-type that is being taken.

2.1.2.5.3.12 get_liveliness_changed_status
This operation allows access to the LIVELINESS_CHANGED communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.

2.1.2.5.3.13 get_requested_deadline_missed_status
This operation allows access to the REQUESTED_DEADLINE_MISSED communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.
2.1.2.5.3.14 get_requested_incompatible_qos_status

This operation allows access to the REQUESTED_INCOMPATIBLE_QOS communication status. Communication statuses are described in Section 2.1.4.1, “Communication Status,” on page 2-76.

2.1.2.5.3.15 get_key_value

This operation can be used to retrieve the instance key that corresponds to an instance_handle. The operation will only fill the fields that form the key inside the key_holder instance.

2.1.2.5.4 DataSample Class

A DataSample represents an atom of data information (i.e., one value for one instance) as returned by DataReader's read/take operations. It consists of two parts: A SampleInfo and the Data.

2.1.2.5.5 SampleInfo Class

<table>
<thead>
<tr>
<th>SampleInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>lifecycle_state</td>
</tr>
<tr>
<td>sample_state</td>
</tr>
<tr>
<td>source_timestamp</td>
</tr>
<tr>
<td>instance_handle</td>
</tr>
</tbody>
</table>

No operations

SampleInfo is the information that accompanies each sample that is 'read' or 'taken'. It contains the following information:

- The lifecycle_state, which can be:
  - NEW if this instance was not previously known by the DataReader;
  - MODIFIED if this instance was previously known by the DataReader, DISPOSED if the instance has been disposed by a DataWriter;
  - NO_WRITERS if none of the DataWriter objects currently “alive” (according to the LIVELINESS QoS) are writing the instance.

Refer to “Access to the data” for a detailed explanation of the lifecycle_state.

- the sample_state (READ or NOT_READ) that indicates whether or not the corresponding data sample has already been read;
- the source_timestamp;
- the instance_handle that identifies locally the corresponding instance.
2.1.2.5.6 SubscriberListener Interface

<table>
<thead>
<tr>
<th>SubscriberListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_data_on_readers</td>
</tr>
<tr>
<td>subscriber Subscriber</td>
</tr>
<tr>
<td>on_sample_lost</td>
</tr>
<tr>
<td>subscriber Subscriber</td>
</tr>
<tr>
<td>status</td>
</tr>
</tbody>
</table>

Since a **Subscriber** is a kind of **Entity**, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type **SubscriberListener**. Its definition can be found in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

2.1.2.5.7 DataReaderListener Interface

<table>
<thead>
<tr>
<th>DataReaderListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_data_available</td>
</tr>
<tr>
<td>the_reader DataReader</td>
</tr>
<tr>
<td>on_sample_rejected</td>
</tr>
<tr>
<td>the_reader DataReader</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>on_liveliness_changed</td>
</tr>
<tr>
<td>the_reader DataReader</td>
</tr>
<tr>
<td>status</td>
</tr>
<tr>
<td>onRequestedDeadlineMissed status RequestedDeadlineMissedStatus</td>
</tr>
<tr>
<td>onRequestedIncompatibleQos status RequestedIncompatibleQosStatus</td>
</tr>
</tbody>
</table>

Since a **DataReader** is a kind of **Entity**, it has the ability to have an associated listener. In this case, the associated listener should be of concrete type **DataReaderListener**. Its definition can be found in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.
2.1.2.5.8 ReadCondition Class

ReadCondition objects are conditions specifically dedicated to read operations and attached to one DataReader.

<table>
<thead>
<tr>
<th>ReadCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>lifecycle_state_mask</td>
</tr>
<tr>
<td>sample_state_mask</td>
</tr>
<tr>
<td>No operations</td>
</tr>
</tbody>
</table>

ReadCondition objects allow an application to specify the data samples it is interested in (by specifying the desired lifecycle-states, as well as sample-states). See the parameter definitions for DataReader’s read/take operations.) This allows the middleware to enable the condition only when suitable information is available. They are to be used in conjunction with a WaitSet as normal conditions. More than one ReadCondition may be attached to the same DataReader.

2.1.2.5.9 QueryCondition Class

QueryCondition objects are specialized ReadCondition objects that allow the application to also specify a filter on the locally available data.

<table>
<thead>
<tr>
<th>QueryCondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>query_expression</td>
</tr>
<tr>
<td>query_arguments</td>
</tr>
<tr>
<td>No operations</td>
</tr>
</tbody>
</table>

The query (query_expression) is similar to a SQL WHERE clause and can be parameterized by arguments that are dynamically changeable by the set_arguments operation.

Precise syntax for the query expression can be found in Appendix A.

This feature is optional (in the cases where it is not supported, the DataReader::create_querycondition should return an error)

22. E.g., the application can specify that the condition must only be enabled when new instances are received by using the NEW lifecycle state.
2.1.3 Supported QoS

The Data-Distribution Service (DDS) relies on the use of QoS. A QoS (Quality of Service) is a set of characteristics that controls some aspect of the behavior of the DDS Service. QoS is comprised of individual QoS policies (objects of type deriving from `QosPolicy`).

![Diagram of Supported QoS policies](image)

QoS (i.e., a list of `QosPolicy` objects) may be associated with all `Entity` objects in the system such as `Topic`, `DataWriter`, `DataReader`, `Publisher`, `Subscriber`, and `DomainParticipant`.

*Figure 2-12  Supported QoS policies*
Some `QosPolicy` values may not be compatible with other ones. These cases are described in the table below. When a set of `QosPolicy` is passed (set_qos operations), the set resulting from adding the new policies on top of the previous is checked for consistency. If the resulting QoS is inconsistent, the change of QoS operation fails and the previous values are retained.

In several cases, for communications to occur properly (or efficiently), a `QosPolicy` on the publisher side must be compatible with a corresponding policy on the subscriber side. For example, if a `Subscriber` requests to receive data reliably while the corresponding `Publisher` defines a best-effort policy, communication will not happen as requested. To address this issue and maintain the desirable de-coupling of publication and subscription as much as possible, the specification for `QosPolicy` follows the subscriber-requested, publisher-offered pattern. In this pattern, the subscriber side can specify a "requested" value for a particular `QosPolicy`. The Publisher side specifies an "offered" value for that `QosPolicy`. The Service will then determine whether the value requested by the subscriber side is compatible with what is offered by the publisher side. If the two policies are compatible then communication will be established. If the two policies are not compatible, the Service will not establish communications between the two `Entity` objects and will record this fact by means of the OFFERED_INCOMPATIBLE_QOS on the publisher end and REQUESTED_INCOMPATIBLE_QOS on the subscriber end (cf. Section 2.1.4.1, “Communication Status,” on page 2-76). The application can detect this fact by means of a listener or conditions (cf. Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75).

The `QosPolicy` objects that need to be set in a compatible manner at the publisher end are indicated by the setting of the ‘RxO23’ property:

- A ‘RxO’ setting of “Yes” indicates that the policy can be set both at the publishing and subscribing ends and the values must be set in a compatible manner. In this case the compatible values are explicitly defined.
- A ‘RxO’ setting of “No” indicates that the policy can be set both at the publishing and subscribing ends but the two settings are independent. That is, all combinations of values are compatible.
- A ‘RxO’ setting of “N/A” indicates that the policy can only be specified at either the publishing or the subscribing end, but not at both ends. So compatibility does not apply.

The ‘changeable’ property determines whether the `QosPolicy` can be changed after the `Entity` is enabled. In other words, a policy with ‘changeable’ setting of ‘NO’ is considered “immutable” and can only be specified either at `Entity` creation time or else prior to calling the `enable` operation on the `Entity`.

The following tables give the list of supported `QosPolicy`: their name, semantics, possible values and the `Entity` they apply to.

---

23.Requested / Offered
<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Value</th>
<th>Meaning</th>
<th>Concerns</th>
<th>RxO</th>
<th>Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_DATA</td>
<td>a sequence of octets</td>
<td>User data not known by the middleware, but distributed by means of built-in topics (cf. Section 2.1.5).</td>
<td>DomainParticipant,</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DataReader, DataWriter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DURABILITY</td>
<td>A “kind”: VOLATILE, TRANSIENT, or PERSISTENT</td>
<td>This policy expresses if the data should 'outlive' their writing time.</td>
<td>Topic, DataReader, DataWriter</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VOLATILE</td>
<td></td>
<td>The Service does not need to keep any samples of data-instances on behalf of any DataReader that is not known by the DataWriter at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSIENT</td>
<td></td>
<td>The Service will attempt to keep some samples so that they can be delivered to any potential late-joining DataReader. Which particular samples are kept depends on other QoS such as HISTORY and RESOURCE_LIMITS. The Service is only required to keep the data in memory and not in permanent storage.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERSISTENT</td>
<td></td>
<td>[optional] Data is kept on permanent storage, so that they can outlive a system session.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESENTATION</td>
<td>An “access_scope”: IND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTANCE</td>
<td></td>
<td>Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application’s ability to: specify and receive coherent changes see the relative order of changes. access_scope determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two booleans control whether coherent access and ordered access are supported within the scope access_scope.</td>
<td>Publisher, Subscriber</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GROUP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTANCE</td>
<td>Scope spans only a single instance. Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QosPolicy</td>
<td>Value</td>
<td>Meaning</td>
<td>Concerns</td>
<td>RxO</td>
<td>Changeable</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>TOPIC</td>
<td>Scope spans to all instances within the same <strong>DataWriter</strong> (or <strong>DataReader</strong>), but not across instances in different <strong>DataWriter</strong> (or <strong>DataReader</strong>).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td>[optional] Scope spans to all instances belonging to <strong>DataWriter</strong> (or <strong>DataReader</strong>) entities within the same <strong>Publisher</strong> (or <strong>Subscriber</strong>).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coherent_access</td>
<td>Specifies support <strong>coherent access</strong>. That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordered_access</td>
<td>Specifies support for <strong>ordered access</strong> to the samples received at the subscription end. That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEADLINE</td>
<td>A duration “deadline_period”</td>
<td><strong>DataReader</strong> expects a new sample updating the value of each instance at least once every <strong>deadline_period</strong>. <strong>DataWriter</strong> indicates that the application commits to write a new value (using the <strong>DataWriter</strong>) for each instance managed by the <strong>DataWriter</strong> at least once every <strong>deadline_period</strong>. The default value of the <strong>deadline_period</strong> is infinite.</td>
<td>Topic, DataReader, DataWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LATENCY_BUDGET</td>
<td>A duration “duration”</td>
<td>Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications. This policy is a hint to the Service, not something that must be monitored or enforced. The Service is not required to track or alert the user of any violation. The default value of the <strong>duration</strong> is zero indicating that the delay should be minimized.</td>
<td>Topic, DataReader, DataWriter</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>A “kind” <strong>SHARED</strong> <strong>EXCLUSIVE</strong></td>
<td>[optional] Specifies whether it is allowed for multiple <strong>DataWriters</strong> to write the same instance of the data and if so, how these modifications should be arbitrated</td>
<td>Topic</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td><strong>SHARED</strong></td>
<td>Indicates shared ownership for each instance. Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an &quot;owner&quot; for the instances. This is the default behavior if the OWNERSHIP QoS policy is not specified or supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QosPolicy</td>
<td>Value</td>
<td>Meaning</td>
<td>Concerns</td>
<td>RxO</td>
<td>Changeable</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td></td>
<td>Indicates each instance can only be owned by one <strong>DataWriter</strong>, but the owner of an instance can change dynamically. The selection of the owner is controlled by the setting of the OWNERSHIP_STRENGTH QoS policy. The owner is always set to be the highest-strength <strong>DataWriter</strong> object among the ones currently “active” (as determined by the LIVELINESS QoS).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OWNERSHIP_STRENGTH</td>
<td>An integer “value”</td>
<td>[optional] Specifies the value of the “strength” used to arbitrate among multiple <strong>DataWriter</strong> objects that attempt to modify the same instance of a data-object (identified by <strong>Topic</strong> + <strong>key</strong>). This policy only applies if the OWNERSHIP QoS policy is of kind EXCLUSIVE.</td>
<td><strong>DataWriter</strong></td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>LIVELINESS</td>
<td>A “kind”: AUTOMATIC, MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC and a duration “lease_duration”</td>
<td>Determines the mechanism and parameters used by the application to determine whether an <strong>Entity</strong> is “active” (alive). The “liveliness” status of an Entity is used to maintain instance ownership in combination with the setting of the OWNERSHIP QoS policy. The application is also informed via listener when an <strong>Entity</strong> is no longer alive. The <strong>DataReader</strong> requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the <strong>lease_duration</strong>. The <strong>DataWriter</strong> commits to signalling its liveliness using the stated means at intervals not to exceed the <strong>lease_duration</strong>. Listeners are used to notify the <strong>DataReader</strong> of loss of liveliness and <strong>DataWriter</strong> of violations to the liveliness contract. The default kind is AUTOMATIC and the default value of the <strong>lease_duration</strong> is infinite.</td>
<td><strong>Topic</strong>, <strong>DataReader</strong>, <strong>DataWriter</strong></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td></td>
<td>The infrastructure will automatically signal liveliness for the <strong>DataWriters</strong> at least as often as required by the <strong>lease_duration</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QosPolicy</td>
<td>Value</td>
<td>Meaning</td>
<td>Concerns</td>
<td>RxO</td>
<td>Changeable</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>MANUAL modes</td>
<td></td>
<td>The user application takes responsibility to signal liveliness to the Service using one of the mechanisms described in Section 2.1.3.7. Liveliness must be asserted at least once every <code>lease_duration</code> otherwise the Service will assume the corresponding <code>Entity</code> or is no longer “active/alive.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MANUAL_BY_PARTICIPANT</td>
<td></td>
<td>The Service will assume that as long as at least one <code>Entity</code> within the domain has asserted its liveliness the <code>Entity</code> is also alive.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MANUAL_BY_TOPIC</td>
<td></td>
<td>The Service will only assume liveliness of the <code>DataWriter</code> if the application has asserted liveliness of that <code>DataWriter</code> itself.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME_BASED_FILTER</td>
<td>A duration &quot;minimum_separation&quot;</td>
<td>Filter that allows a <code>DataReader</code> to specify that it interested only in (potentially) a subset of the values of the data. The filter states that the <code>DataReader</code> does not want to receive more than one value each <code>minimum_separation</code> regardless of how fast the changes occur. The setting of this QoS policy is incompatible with RELIABILITY policy set to ALL. By default <code>minimum_separation</code>=0 indicating <code>DataReader</code> is potentially interested in all values.</td>
<td><code>DataReader</code></td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>PARTITION</td>
<td>A list of strings “names”</td>
<td>Set of strings that introduces a logical partition among the topics visible by the <code>Publisher</code> and <code>Subscriber</code>. A <code>DataWriter</code> within a <code>Publisher</code> only communicates with a <code>DataReader</code> in a <code>Subscriber</code> if (in addition to matching the <code>Topic</code> and having compatible QoS) the <code>Publisher</code> and <code>Subscriber</code> have a common partition name string.</td>
<td><code>Publisher</code>, <code>Subscriber</code></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>A “kind”: RELIABLE, BEST_EFFORT</td>
<td>Indicates the level of reliability offered by the Service.</td>
<td><code>Topic</code>, <code>DataReader</code>, <code>DataWriter</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>QosPolicy</td>
<td>Value</td>
<td>Meaning</td>
<td>Concerns</td>
<td>RxO</td>
<td>Changeable</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>RELIABLE</td>
<td>Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the DataWriter) the middleware guarantees that all samples in the DataWriter history will eventually be delivered to the allDataReader objects. Outside steady state the HISTORY and RESOURCE_LIMITS policies will determine how samples become part of the history and whether a samples can be discarded from it.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEST_EFFORT</td>
<td>Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples. This is the default value.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESTINATION_ORDER</td>
<td>A “kind”:</td>
<td>Controls the criteria used to determine the logical order among changes made by Publisher entities to the same instance of data (i.e., matching Topic and key).</td>
<td>Topic,</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>BY_RECEPTION_TIMESTAMP,</td>
<td></td>
<td>DataReader</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY_SOURCE_TIMESTAMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY_RECEPTION_TIMESTAMP</td>
<td>Indicates that data is ordered based on the reception time at each Subscriber. Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY_SOURCE_TIMESTAMP</td>
<td>Indicates that data is ordered based on a time-stamp placed at the source (by the Service or by the application). In any case this guarantees a consistent final value for the data in all subscribers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QoSPolicy</td>
<td>Value</td>
<td>Meaning</td>
<td>Concerns</td>
<td>RxO</td>
<td>Changeable</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>HISTORY</td>
<td>A “kind”: KEEP_LAST, KEEP_ALL And an optional integer “depth”</td>
<td>Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the <strong>DataWriter</strong> on behalf of existing <strong>DataReader</strong> entities. The behavior with regards to a <strong>DataReader</strong> entities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application “takes” them from the Service.</td>
<td>Topic, DataReader, DataWriter</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>KEEP_LAST</td>
<td>KEEP_LAST and optional integer “depth”</td>
<td>On the publishing side, the Service will only attempt to keep the most recent “depth” samples of each instance of data (identified by its key) managed by the <strong>DataWriter</strong>. On the subscribing side, the <strong>DataReader</strong> will only attempt to keep the most recent “depth” samples received for each instance (identified by its key) until the application “takes” them via the <strong>DataReader</strong>’s <strong>take</strong> operation. KEEP_LAST is the default kind. The default value of <strong>depth</strong> is 1. If a value other than 1 is specified, it should be compatible with the settings of the RESOURCE_LIMITS QoS policy.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEEP_ALL</td>
<td>KEEP_ALL</td>
<td>On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its key) managed by the <strong>DataWriter</strong> until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its key) managed by the <strong>DataReader</strong>. These samples are kept until the application “takes” them from the Service via the <strong>take</strong> operation. The setting of <strong>depth</strong> has no effect. Its implied value is “INFINITE”(^2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.1.3.1 USER_DATA

The purpose of this QoS is to allow the application to attach additional information to the created Entity objects such that when a remote application discovers their existence it can access that information and use it for its own purposes. One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source. In combination with operations such as ignore_participant, ignore_publication, ignore_subscription and ignore_topic, these QoS can assist an application to define and enforce its own security policies. The use of this QoS is not limited to security, rather it offers a simple, yet flexible extensibility mechanism.

2.1.3.2 DURABILITY

The decoupling between DataReader and DataWriter offered by the Publish/Subscribe paradigm allows an application to write data even if there are no current readers on the network. Moreover, a DataReader that joins the network after some data has been written could potentially be interested in accessing the most current values of the data as well as potentially some history. This QoS policy controls whether the Service will actually make data available to late-joining readers. Note that although related, this does...
not strictly control what data the Service will maintain internally. That is, the Service may choose to maintain some data for its own purposes (e.g. flow control) and yet not make it available to late-joining readers if the DURABILITY QoS policy is set to VOLATILE.

The value offered is considered compatible with the value requested if and only if the inequality “offered kind > requested kind” evaluates to ‘TRUE’. For the purposes of this inequality, the values of DURABILITY kind are considered ordered such that VOLATILE < TRANSIENT < PERSISTENT.

### 2.1.3.3 PRESENTATION

This QoS policy controls the extent to which changes to data-instances can be made dependent on each other and also the kind of dependencies that can be propagated and maintained by the Service.

The setting of coherent_access controls whether the Service will preserve the groupings of changes made by the publishing application by means of the operations begin_coherent_change and end_coherent_change.

The setting of ordered_access controls whether the Service will preserve the order of changes.

The granularity is controlled by the setting of the access_scope.

If coherent_access is set, then the access_scope controls the maximum extent of coherent changes. The behavior is as follows:

- If access_scope is set to INSTANCE, the use of begin_coherent_change and end_coherent_change has no effect on how the subscriber can access the data because with the scope limited to each instance, changes to separate instances are considered independent and thus cannot be grouped by a coherent change.

- If access_scope is set to TOPIC, then coherent changes (indicated by their enclosure within calls to begin_coherent_change and end_coherent_change) will be made available as such to each remote DataReader independently. That is, changes made to instances within the each individual DataWriter will be available as coherent with respect to other changes to instances in that same DataWriter, but will not be grouped with changes made to instances belonging to a different DataWriter.

- If access_scope is set to GROUP, then coherent changes made to instances through a DataWriter attached to a common Publisher are made available as a unit to remote subscribers.

If ordered_access is set, then the access_scope controls the maximum extent for which order will be preserved by the Service/

- If access_scope is set to INSTANCE (the lowest level), then changes to each instance are considered unordered relative to changes to any other instance. That means that changes (creations, deletions, modifications) made to two instances are not necessarily seen in the order they occur. This is the case even if it is the same application thread making the changes using the same DataWriter.
• If access_scope is set to TOPIC, changes (creations, deletions, modifications) made by a single DataWriter are made available to subscribers in the same order they occur. Changes made to instances though different DataWriter entities are not necessarily seen in the order they occur. This is the case, even if the changes are made by a single application thread using DataWriter objects attached to the same Publisher.

• Finally, if access_scope is set to GROUP, changes made to instances via DataWriter entities attached to the same Publisher object are made available to subscribers on the same order they occur.

Note that this QoS policy controls the scope at which related changes are made available to the subscriber. This means the subscriber can access the changes in a coherent manner and in the proper order; however, it does not necessarily imply that the Subscriber will indeed access the changes in the correct order. For that to occur, the application at the subscriber end must use the proper logic in reading the DataReader objects, as described in “Access to the data.”

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. The inequality “offered access_scope > requested access_scope” evaluates to ‘TRUE’. For the purposes of this inequality, the values of PRESENTATION access_scope are considered ordered such that INSTANCE < TOPIC < GROUP.

2. Requested coherent_access is FALSE, or else both offered and requested coherent_access are TRUE.

3. Requested ordered_access is FALSE, or else both offered and requested ordered _access are TRUE.

2.1.3.4 DEADLINE

This policy is useful for cases where a Topic is expected to have each instance updated periodically. On the publishing side this setting establishes a contract that the application must meet. On the subscribing side the setting establishes a minimum requirement for the remote publishers that are expected to supply the data values.

When the Service ‘matches’ a DataWriter and a DataReader it checks whether the settings are compatible (i.e., offered deadline <= requested deadline) if they are not, the two entities are informed (via the listener or condition mechanism) of the incompatibility of the QoS settings and communication will not occur.

Assuming that the reader and writer ends have compatible settings, the fulfilment of this contract is monitored by the Service and the application is informed of any violations by means of the proper listener or condition.

The value offered is considered compatible with the value requested if and only if the inequality “offered period <= requested period” evaluates to ‘TRUE’.
2.1.3.5 LATENCY_BUDGET

This policy provides a means for the application to indicate to the middleware the “urgency” of the data-communication. By having a non-zero duration the Service can optimize its internal operation.

This policy is considered a hint. Therefore the Service will not fail to match a DataReader with a DataWriter due to incompatibility on this QoS, rather it will automatically adapt its behavior on the publishing end to meet the requirements of all subscribers. Consequently this QoS will never trigger an incompatible QoS notification, nor does it have any listeners associated with violations of the contract.

The value offered is considered compatible with the value requested if and only if the inequality “offered duration <= requested duration” evaluates to ‘TRUE’.

2.1.3.6 OWNERSHIP

This policy controls whether the Service allows multiple DataWriter objects to update the same instance (identified by Topic + key) of a data-object.

This QoS policy only applies to Topic and not to DataReader or DataWriter. The reason for this is that it would make no sense for a DataReader or a DataWriter to override the setting in the Topic.

There are two kinds of OWNERSHIP selected by the setting of the kind: SHARED and EXCLUSIVE.

2.1.3.6.1 SHARED kind

This setting indicates that the Service does not enforce unique ownership for each instance. In this case, multiple writers can update the same data-object instance. The subscriber to the Topic will be able to access modifications from all DataWriter objects, subject to the settings of other QoS that may filter particular samples (e.g. the TIME_BASED_FILTER or HISTORY QoS policy). In any case there is no “filtering” of modifications made based on the identity of the DataWriter that causes the modification.

2.1.3.6.2 EXCLUSIVE kind

This setting indicates that each instance of a data-object can only be modified by one DataWriter. In other words, at any point in time a single DataWriter “owns” each instance and is the only one whose modifications will be visible to the DataReader objects. The owner is determined by selecting the DataWriter with the highest value of the strength\(^{24}\) that is currently “alive” as defined by the LIVELINESS QoS policy. Ownership can therefore change as a result of (a) a DataWriter in the system with a higher value of the strength that modifies the instance, (b) a change in the strength value of the DataWriter that owns the instance, and (c) a change in the liveliness of the DataWriter that owns the instance.

\(^{24}\)The “strength” of a DataWriter is the value of its OWNERSHIP_STRENGTH QoS.
The behavior of the system is as if the determination was made independently by each DataReader. Each DataReader may detect the change of ownership at a different time. It is not a requirement that at a particular point in time all the DataReader objects for that Topic have a consistent picture of who owns each instance.

It is also not a requirement that the DataWriter objects are aware of whether they own a particular instance. There is no error or notification given to a DataWriter that modifies an instance it does not currently own.

The requirements are chosen to (a) preserve the decoupling of publishers and subscriber, and (b) allow the policy to be implemented efficiently.

It is possible that multiple DataWriter objects with the same strength modify the same instance. If this occurs the Service will pick one of the DataWriter objects as the “owner”. It is not specified how the owner is selected. However, it is required that the policy used to select the owner is such that all DataReader objects will make the same choice of the particular DataWriter that is the owner. It is also required that the owner remains the same until there is a change in strength, liveliness, or a new DataWriter with higher strength modifies the instance. The DataReader is also notified of this via a status change that is accessible by means of the Listener or Condition mechanisms.

Exclusive ownership is on an instance-by-instance basis. That is, a subscriber can receive values written by a lower strength DataWriter as long as they affect instances whose values have not been set by the higher-strength DataWriter.

The value of the OWNERSHIP kind offered must exactly match the one requested or else they are considered incompatible.

2.1.3.7 LIVELINESS

This policy controls the mechanism and parameters used by the Service to ensure that particular entities on the network are still “alive”. The liveliness can also affect the ownership of a particular instance, as determined by the OWNERSHIP QoS policy.

This policy has several settings to support both data-objects that are updated periodically as well as those that are changed sporadically. It also allows customizing for different application requirements in terms of the kinds of failures that will be detected by the liveliness mechanism.

The AUTOMATIC liveliness setting is most appropriate for applications that only need to detect failures at the process-level, but not application-logic failures within a process. The Service takes responsibility for renewing the leases at the required rates and thus, as long as the local process where a DomainParticipant is running and the link connecting it to remote participants remains connected, the entities within the DomainParticipant will be considered alive. This requires the lowest overhead.

25 Process here is used to mean an operating system-process as in an address space providing the context where a number of threads execute.
The MANUAL settings (MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC), require the application on the publishing side to periodically assert the liveliness before the lease expires to indicate the corresponding Entity is still alive. The action can be explicit by calling the assert_liveliness operations, or implicit by writing some data.

The two possible manual settings control the granularity at which the application must assert liveliness.

- The setting MANUAL_BY_PARTICIPANT requires only that one Entity within the publisher is asserted to be alive to deduce all other Entity objects within the same DomainParticipant are also alive.

- The setting MANUAL_BY_TOPIC requires that at least one instance within the DataWriter is asserted.

The value offered is considered compatible with the value requested if and only if the following conditions are met:

1. the inequality “offered kind > requested kind” evaluates to ‘TRUE’. For the purposes of this inequality, the values of LIVELINESS kind are considered ordered such that: AUTOMATIC < MANUAL_BY_PARTICIPANT < MANUAL_BY_TOPIC.

2. the inequality “offered lease_duration < requested lease_duration” evaluates to TRUE.

2.1.3.8 TIME_BASED_FILTER

This policy allows a DataReader to indicate that it does not necessarily want to see all values of each instance published under the Topic. Rather, it wants to see at most one change every minimum_separation period.

This setting allows a DataReader to further decouple itself from the DataWriter objects. It can be used to protect applications that are running on a heterogeneous network where some nodes are capable of generating data much faster than others can consume it. It also accommodates the fact that for fast-changing data different subscribers may have different requirements as to how frequently they need to be notified of the most current values.

2.1.3.9 PARTITION

This policy allows the introduction of a logical partition concept inside the 'physical' partition induced by a domain.

For a DataReader to see the changes made to an instance by a DataWriter, not only the Topic must match, but also they must share a common partition. Each string in the list that defines this QoS policy defines a partition name. A partition name may contain wildcards. Sharing a common partition means that one of the partition names matches.

Failure to match partitions is not considered an “incompatible” QoS and does not trigger any listeners nor conditions.
By default, **DataWriter** and **DataReader** objects belonging to **Publisher** or **Subscriber** that do not specify a PARTITION policy will participate in the default partition (whose name is "").

Partitions are different from creating **Entity** objects in different domains in several ways. First, entities belonging to different domains are completely isolated from each other; there is no traffic, meta-traffic or any other way for an application or the Service itself to see entities in a domain it does not belong to. Second, an **Entity** can only belong to one domain whereas an **Entity** can be in multiple partitions. Finally, as far as the DDS Service is concerned, each unique data instance is identified by the tuple (domainId, **Topic**, key). Therefore two **Entity** objects in different domains cannot refer to the same data instance. On the other hand, the same data-instance can be made available (published) or requested (subscribed) on one or more partitions.

### 2.1.3.10 RELIABILITY

This policy indicates the level of reliability requested by a **DataReader** or offered by a **DataWriter**. These levels are ordered, BEST_EFFORT being lower than RELIABLE. A **DataWriter** offering a level is implicitly offering all levels below.

The setting of this policy has a dependency on the setting of the HISTORY and RESOURCE_LIMITS policies. In case the RELIABILITY kind is set to RELIABLE and the HISTORY kind set to KEEP_ALL the write operation on the **DataWriter** may block if the modification would cause data to be lost or else cause one of the limits in specified in the RESOURCE_LIMITS to be exceeded.

The value offered is considered compatible with the value requested if and only if the inequality ‘offered kind >= requested kind’ evaluates to ‘TRUE’. For the purposes of this inequality, the values of RELIABILITY kind are considered ordered such that BEST_EFFORT < RELIABLE.

### 2.1.3.11 DESTINATION_ORDER

This policy controls how each subscriber resolves the final value of a data instance that is written by multiple **DataWriter** objects (which may be associated with different **Publisher** objects) running on different nodes.

The setting BY_RECEPTION_TIMESTAMP indicates that, assuming the STRENGTH policy allows it, the latest received value for the instance should be the one whose value is kept. This is the default value.

The setting BY_SOURCE_TIMESTAMP indicates that, assuming the STRENGTH policy allows it, a timestamp placed at the source should be used. This is the only setting that, in the case of concurrent same-strength **DataWriter** objects updating the same instance, ensures all subscribers will end up with the same final value for the instance.

The mechanism to set the source timestamp is middleware dependent.
2.1.3.12 HISTORY

1. This policy controls the behavior of the Service when the value of an instance changes before it is finally communicated to some of its existing *DataReader* entities.

2. If the *kind* is set to KEEP_LAST, then the Service will only attempt to keep the latest values of the instance and discard the older ones. In this case, the value of *depth* regulates the maximum number of values (up to and including the most current one) the Service will maintain and deliver. The default (and most common setting) for *depth* is one, indicating that only the most recent value should be delivered.

3. If the *kind* is set to KEEP_ALL, then the Service will attempt to maintain and deliver all the values of the instance to existing subscribers. The resources that the Service can use to keep this history are limited by the settings of the RESOURCE_LIMITS QoS. If the limit is reached, then the behavior of the Service will depend on the RELIABILITY QoS. If the reliability kind is BEST_EFFORT, then the old values will be discarded. If reliability is RELIABLE, then the Service will block the *DataWriter* until it can deliver the necessary old values to all subscribers.

2.1.3.13 RESOURCE_LIMITS

This policy controls the resources that the Service can use in order to meet the requirements imposed by the application and other QoS settings.

If the *DataWriter* objects are communicating samples faster than they are ultimately taken by the *DataReader* objects, the middleware will eventually hit against some of the QoS-imposed resource limits. Note that this may occur when just a single *DataReader* cannot keep up with its corresponding *DataWriter*. The behavior in this case depends on the setting for the RELIABILITY QoS. If reliability is BEST_EFFORT then the Service is allowed to drop samples. If the reliability is RELIABLE, the Service will block the *DataWriter* or discard the sample at the *DataReader* in order not to lose existing samples.

2.1.4 Listeners, Conditions and Wait-sets

Listeners and conditions (in conjunction with wait-sets) are two alternative mechanisms that allow the application to be made aware of changes in the DCPS communication status.

26. So that the sample can be re-sent at a later time.
### 2.1.4.1 Communication Status

The communication statuses whose changes can be communicated to the application depend on the *Entity*. The following table shows for each entity the statuses that are relevant.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Status Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>INCONSISTENT_TOPIC</td>
<td>Another topic exists with the same name, but different characteristics.</td>
</tr>
<tr>
<td>Subscriber</td>
<td>SAMPLE_LOST</td>
<td>A sample has been lost (never received).</td>
</tr>
<tr>
<td></td>
<td>DATA_ON_READERS</td>
<td>New information is available.</td>
</tr>
<tr>
<td>DataReader</td>
<td>SAMPLE_REJECTED</td>
<td>A (received) sample has been rejected.</td>
</tr>
<tr>
<td></td>
<td>LIVELINESS_CHANGED</td>
<td>The liveness of one or more DataWriter that were writing instances read through the DataReader has changed. Some DataWriter have become “active” or “inactive”.</td>
</tr>
<tr>
<td></td>
<td>REQUESTED_DEADLINE_MISSED</td>
<td>The deadline that the DataReader was expecting through its QosPolicy DEADLINE was not respected for a specific instance.</td>
</tr>
<tr>
<td></td>
<td>REQUESTED_INCOMPATIBLE_QOS</td>
<td>A QosPolicy value was incompatible with what is offered.</td>
</tr>
<tr>
<td></td>
<td>DATAAVAILABLE</td>
<td>New information is available.</td>
</tr>
<tr>
<td>DataWriter</td>
<td>LIVELINESS_LOST</td>
<td>The liveness that the DataWriter has committed through its QosPolicy LIVELINESS was not respected, thus DataReader entities will consider the DataWriter as no longer “active”.</td>
</tr>
<tr>
<td></td>
<td>OFFERED_DEADLINE_MISSED</td>
<td>The deadline that the DataWriter has committed through its QosPolicy DEADLINE was not respected for a specific instance.</td>
</tr>
<tr>
<td></td>
<td>OFFERED_INCOMPATIBLE_QOS</td>
<td>A QosPolicy value was incompatible with what was requested.</td>
</tr>
</tbody>
</table>

Those statuses may be classified in:

- read communication statuses: i.e., those that are related to arrival of data, namely DATA_ON_READERS and DATA_AVAILABLE;
- plain communication statuses: i.e., all the others.

Read communication statuses are treated slightly differently than the others for they don’t change independently. In other words, at least two changes will appear at the same time (DATA_ON_READERS + DATA_AVAILABLE) and even several of the last kind may be part of the set. This ‘grouping’ has to be communicated to the application. How this is done is discussed in each of the two following sections.

For each plain communication status, there is a corresponding structure to hold the status value. These values contain the information related to the change of status, as well as information related to the statuses themselves (e.g., contains cumulative counts). They are used with the two different mechanisms explained in the following sections.
The interpretation of the attributes for each status value is provided in the following table.
<table>
<thead>
<tr>
<th>Status Type</th>
<th>Attribute Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleLostStatus</td>
<td>total_count: Total cumulative count of all samples lost across instances of topics</td>
</tr>
<tr>
<td></td>
<td>subscribed by the Subscriber</td>
</tr>
<tr>
<td></td>
<td>total_count_change: The incremental number of samples lost since the last time the</td>
</tr>
<tr>
<td></td>
<td>listener was called or the status was read</td>
</tr>
<tr>
<td>SampleRejectedStatus</td>
<td>total_count: Total cumulative count of samples rejected by the DataReader</td>
</tr>
<tr>
<td></td>
<td>total_count_change: The incremental number of samples rejected since the last time</td>
</tr>
<tr>
<td></td>
<td>the listener was called or the status was read</td>
</tr>
<tr>
<td></td>
<td>last_reason: Reason for rejecting the last sample rejected.</td>
</tr>
<tr>
<td></td>
<td>last_instance_handle: Handle to the instance being updated by the last sample that</td>
</tr>
<tr>
<td></td>
<td>was rejected.</td>
</tr>
<tr>
<td>InconsistentTopicStatus</td>
<td>total_count: Total cumulative count of the Topics discovered whose name matches</td>
</tr>
<tr>
<td></td>
<td>the Topic to which this status is attached and whose type is inconsistent with the</td>
</tr>
<tr>
<td></td>
<td>Topic.</td>
</tr>
<tr>
<td></td>
<td>total_count_change: The type of the last Topic discovered that had the same name as</td>
</tr>
<tr>
<td></td>
<td>the Topic to which this status is attached but had an inconsistent type.</td>
</tr>
<tr>
<td>LivelinessChangedStatus</td>
<td>active_count: The total count of currently active DataWriters that write the Topic</td>
</tr>
<tr>
<td></td>
<td>the DataReader reads.</td>
</tr>
<tr>
<td></td>
<td>inactive_count: The total count of currently inactive DataWriters that write the</td>
</tr>
<tr>
<td></td>
<td>Topic the DataReader reads.</td>
</tr>
<tr>
<td></td>
<td>active_count_change: The change in the active_count since the last time the listener</td>
</tr>
<tr>
<td></td>
<td>was called or the status was read.</td>
</tr>
<tr>
<td></td>
<td>inactive_count_change: The change in the inactive_count since the last time the</td>
</tr>
<tr>
<td></td>
<td>listener was called or the status was read.</td>
</tr>
<tr>
<td>RequestedDeadlineMissedStatus</td>
<td>total_count: Total cumulative count of the deadlines detected for any instance read</td>
</tr>
<tr>
<td></td>
<td>by the DataReader.</td>
</tr>
<tr>
<td></td>
<td>total_count_change: The incremental number of deadlines detected since the last time</td>
</tr>
<tr>
<td></td>
<td>the listener was called or the status was read.</td>
</tr>
<tr>
<td></td>
<td>last_instance_handle: Handle to the last instance in the DataReader for which a</td>
</tr>
<tr>
<td></td>
<td>deadline was detected.</td>
</tr>
<tr>
<td>RequestedIncompatibleQosStatus</td>
<td>total_count: Total cumulative count the concerned DataReader discovered a DataWriter</td>
</tr>
<tr>
<td></td>
<td>for the same Topic with an offered QoS that is incompatible with that requested by</td>
</tr>
</tbody>
</table>
2.1.4.2 Changes in Status

Associated with each one of an Entity’s communication status is a logical StatusChangedFlag. This flag indicates whether that particular communication status has changed since the last time the status was ‘read’ by the application. The way the status changes is slightly different for the Plain Communication Status and the Read Communication status.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_policy_id</td>
<td>The PolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.</td>
</tr>
<tr>
<td>policies</td>
<td>A list containing for each policy the total number of times that the concerned DataReader discovered a DataWriter for the same Topic with an offered QoS that is incompatible with that requested by the DataReader.</td>
</tr>
<tr>
<td>LivelinessLostStatus</td>
<td>Attribute meaning</td>
</tr>
<tr>
<td>total_count</td>
<td>Total cumulative count of the number of times the DataWriter failed to actively signal its liveness within the offered liveness period.</td>
</tr>
<tr>
<td>total_count_change</td>
<td>The change in total_count since the last time the listener was called or the status was read.</td>
</tr>
<tr>
<td>OfferedDeadlineMissedStatus</td>
<td>Attribute meaning</td>
</tr>
<tr>
<td>total_count</td>
<td>Total cumulative count of the number of times the DataWriter failed to write within its offered deadline.</td>
</tr>
<tr>
<td>total_count_change</td>
<td>The change in total_count since the last time the listener was called or the status was read.</td>
</tr>
<tr>
<td>last_instance_handle</td>
<td>Handle to the last instance in the DataWriter for which an offered deadline was missed.</td>
</tr>
<tr>
<td>OfferedIncompatibleQosStatus</td>
<td>Attribute meaning</td>
</tr>
<tr>
<td>total_count</td>
<td>Total cumulative count the concerned DataWriter discovered a DataReader for the same Topic with a requested QoS that is incompatible with that offered by the DataWriter.</td>
</tr>
<tr>
<td>total_count_change</td>
<td>The change in total_count since the last time the listener was called or the status was read.</td>
</tr>
<tr>
<td>last_policy_id</td>
<td>The PolicyId_t of one of the policies that was found to be incompatible the last time an incompatibility was detected.</td>
</tr>
<tr>
<td>policies</td>
<td>A list containing for each policy the total number of times that the concerned DataWriter discovered a DataReader for the same Topic with a requested QoS that is incompatible with that offered by the DataWriter.</td>
</tr>
</tbody>
</table>
2.1.4.2.1 Changes in Plain Communication Status

For the plain communication status, the StatusChangedFlag flag is initially set to FALSE. It becomes TRUE whenever the plain communication status changes and it is reset to FALSE each time the application accesses the plain communication status via the proper get_<plain communication status> operation on the Entity.

For example, the value of the StatusChangedFlag associated with the REQUESTED_DEADLINE_MISSED status will become TRUE each time new deadline occurs (which increases the total_count field within RequestedDeadlineMissedStatus). The value changes to FALSE when the application accesses the status via the corresponding getRequestedDeadlineMissedStatus method on the proper Entity.
2.1.4.2.2 Changes in Read Communication Statuses

For the read communication status, the StatusChangedFlag flag is initially set to FALSE. It becomes TRUE when data arrives and it is reset to FALSE when all the data is removed from the responsibility of the middleware via the take operation on the proper DataReader entities.

![Diagram](data.png)

**Figure 2-16** Changes in StatusChangedFlag for read communication status

2.1.4.3 Access through Listeners

Listeners provide a mechanism for the middleware to asynchronously alert the application of the occurrence of relevant status changes.

All Entity support a listener, which type of which is specialized to the specific type of the related Entity (e.g., DataReaderListener for the DataReader). Listeners are interfaces that the application must implement. Each dedicated listener presents a list of operations that correspond to the relevant communication status changes (i.e., that the application may react to).

All listeners are listed on Figure 17, associated with the DCPS constructs that participate in this mechanism (note that only the related operations are displayed).
Listeners are stateless. It is thus possible to share the same `DataReaderListener` instance among all the `DataReader` objects (assuming that they will react similarly on similar status changes). Consequently, the provided parameter contains a reference to the actual concerned `Entity`.

### 2.1.4.3.1 Listener Access to Plain Communication Status

The general mapping between the plain communication statuses as explained in Section 2.1.4.1, “Communication Status,” on page 2-76 and the listeners’ operations is as follows:

- For each communication status, there is a corresponding operation whose name is `on_<communication_status>`, which takes a parameter of type `<communication_status>` as listed in Section 2.1.4.1, “Communication Status,” on page 2-76.
- `on_<communication_status>` is available on the relevant `Entity` as well as those that embed it, as expressed on the following figure:
When the application attaches a listener on an entity, it must set a mask that indicates to the middleware which operations are enabled within this listener (cf. operation `Entity::set_listener`).

When a plain communication status changes\(^{27}\), the middleware triggers the most 'specific' relevant listener operation that is enabled\(^{28}\).

This behavior allows the application to set a default behavior (e.g., in the listener associated with the `DomainParticipant`) and setting dedicated behaviors only where needed.

### 2.1.4.3.2 Listener access to Read Communication Status

The two statuses related to data arrival are treated slightly differently. Since they constitute the real purpose of the Data Distribution Service, there is not really a need to provide a default mechanism as for the plain communication statuses, and more importantly, several of them may need to be treated as a whole as explained in Section 2.1.4.1, “Communication Status,” on page 2-76.

The rule is as follows. Each time the read communication status changes\(^{29}\):

- first, the middleware tries to trigger the `SubscriberListener` operation `on_data_on_readers` with a parameter of the related `Subscriber`;
- if this does not succeed (no listener or operation non-enabled) it tries to trigger `on_data_available` on all the related `DataReaderListener` objects, with as parameter the related `DataReader`.

---

27. To be more precise, when the corresponding `StatusChangedFlag` described in “Changes in Plain Communication Status” becomes TRUE.

28. E.g., in case of ON_OFFERED_DEADLINE_MISSED for a given `DataWriter`: the `DataWriter`’s listener operation `on_offered_deadline_missed`, or by default (i.e. if there was no listener attached to that `DataWriter`, or if the operation was not enabled), the `Publisher`’s listener or else (no listener attached to the `Publisher` or operation not enabled) the `DomainParticipant`’s listener.

29. To be more precise, when the corresponding `StatusChangedFlag` described in “Changes in Read Communication Statuses” becomes TRUE.
The rationale is that either the application is interested in relations among data arrivals and it must use the first option (and then get the corresponding `DataReader` objects by calling `get_datareaders` on the related `Subscriber` and then get the data by calling `read/take` on the returned `DataReader` objects\(^{30}\), or it wants to treat each `DataReader` fully independently and it may choose the second option (and then get the data by calling `read/take` on the related `DataReader`).

Note that if `on_data_on_readers` is called, then the middleware will not try to call `on_data_available`, however, the application can force a call to the `DataReader` objects that have data by means of the `notify_datareaders` operation.

### 2.1.4.4 Conditions and Wait-sets

As previously mentioned, conditions (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to communicate communication status changes (including arrival of data) to the application.

Figure 19: Wait-sets and Conditions shows all the DCPS constructs that are involved in that mechanism (note that only the related operations are displayed).

---

\(^{30}\)As detailed in Section 2.1.2.5, “Subscription Module,” on page 2-43.
This mechanism is wait-based. Its general use pattern is as follows:

- The application indicates which relevant information it wants to get, by creating `Condition` objects (StatusCondition, ReadCondition or QueryCondition) and attaching them to a `WaitSet`.

• It then waits on that \texttt{WaitSet} until the \texttt{trigger\_value} of one or several \texttt{Condition} objects become \texttt{TRUE}.

• It then uses the result of the \texttt{wait} (i.e., the list of \texttt{Condition} objects with \texttt{trigger\_value==TRUE}) to actually get the information:
  • by calling \texttt{get\_status\_changes} and then \texttt{get\_<communication\_status>} on the relevant \texttt{Entity}, if the condition is a \texttt{StatusCondition} and the status changes refer to plain communication status;
  • by calling \texttt{get\_status\_changes} and then \texttt{get\_datareaders} on the relevant \texttt{Subscriber} if the condition is a \texttt{StatusCondition} and the status changes refers to \texttt{DATA\_ON\_READERS\^{32}};
  • by calling \texttt{get\_status\_changes} and then \texttt{read/take} on the relevant \texttt{DataReader} if the condition is a \texttt{StatusCondition} and the status changes refers to \texttt{DATA\_AVAILABLE};
  • by calling directly \texttt{read\_w\_condition/take\_w\_condition} on the \texttt{DataReader} with the \texttt{Condition} as a parameter if it is a \texttt{ReadCondition} or a \texttt{QueryCondition}.

Usually the first step is done in an initial is at i on phase, while the others are put in the application main loop.

As there is no extra information passed from the middleware to the application when a wait returns (only the list of triggered \texttt{Condition} objects), \texttt{Condition} objects are meant to embed all that is needed to react properly when enabled. In particular, \texttt{Entity}-related conditions\textsuperscript{33} are related to exactly one \texttt{Entity} and cannot be shared.

The blocking behavior of the \texttt{WaitSet} is illustrated in Figure 19. The result of a \texttt{wait} operation depends on the state of the \texttt{WaitSet}, which in turn depends on whether at least one attached \texttt{Condition} has a \texttt{trigger\_value} of \texttt{TRUE}. If the \texttt{wait} operation is called on a \texttt{WaitSet} with state \texttt{BLOCKED} it will block the calling thread. If \texttt{wait} is called on a \texttt{WaitSet} with state \texttt{UNBLOCKED} it will return immediately. In addition, when the \texttt{WaitSet} transitions from \texttt{BLOCKED} to \texttt{UNBLOCKED} it wakes up any threads that had called \texttt{wait} on it.

\textsuperscript{32}And then read/take on the returned \texttt{DataReader} objects.

\textsuperscript{33}I.e. \texttt{StatusCondition}, \texttt{ReadCondition} and \texttt{FilteredCondition}. Cf. Section 2.1.2.1, “Infra-structure Module,” on page 2-10 on the use of basic Condition.
A key aspect of the Condition/WaitSet mechanism is the setting of the trigger_value of each Condition.

### 2.1.4.4.1 Trigger State of the StatusCondition

The trigger_value of a StatusCondition is the Boolean OR of the ChangedStatusFlag of all the communication statuses to which it is sensitive. That is, trigger_value==FALSE only if all the values of the ChangedStatusFlags are FALSE.

The sensitivity of the StatusCondition to a particular communication status is controlled by the list of enabled_statuses set on the condition by means of the set_enabled_statuses operation.

### 2.1.4.4.2 Trigger State of the ReadCondition

Similar to the StatusCondition, a ReadCondition also has a trigger_value that determines whether the attached WaitSet is BLOCKED or UNBLOCKED. However, unlike the StatusCondition, the trigger_value of the ReadCondition is tied to the presence of at least a sample managed by the Service with LifecycleState and SampleState matching those of the ReadCondition. Furthermore, for the QueryCondition to have a trigger_value==TRUE, the data associated with the sample must be such that the query_expression evaluates to TRUE.

To elaborate further, consider the following example: A ReadCondition that has a lifecycle_state_mask = {NEW}, and sample_state_mask = {NOT_READ}, will have trigger_value of TRUE whenever a new sample arrives and will transition to FALSE as soon as all the NEW samples are either read (so their status changes to READ) or taken (so they are no longer managed by the Service). However if the same ReadCondition had a sample_state_mask = {READ, NOT_READ}, then the trigger_value would only become FALSE once all the new samples are taken (it is not sufficient to read them as that would only change the SampleState to READ but the sample would still have (LifecycleState, SampleState) = (NEW, READ) which overlaps the mask on the ReadCondition.
2.1.4.3 Trigger State of the GuardCondition

The trigger_value of a GuardCondition is completely controlled by the application via operation set_trigger_value.

2.1.4.5 Combination

Those two mechanisms may be combined in the application (e.g., using wait-sets and conditions to access the data and listeners to be warned asynchronously of erroneous communication statuses).

It is likely that the application will choose one or the other mechanism for each particular communication status (not both). However, if both mechanisms are enabled, then the listener mechanism is used first and then the WaitSet objects are signalled.

2.1.5 Built-in Topics

As part of its operation, the middleware must discover and possibly keep track of the presence of remote entities such as a new participant in the domain. This information may also be important to the application, which may want to react to this discovery, or else access it on demand.

To make this information accessible to the application, the DCPS specification introduces a set of built-in topics and corresponding DataReader objects that can then be used by the application. The information is then accessed as if it was normal application data. This approach avoids introducing a new API to access this information and allows the application to become aware of any changes in those values by means of any of the mechanisms presented in Section 2.1.4, “Listeners, Conditions and Wait-sets,” on page 2-75.

The built-in data-readers all belong to a built-in Subscriber. This subscriber can be retrieved by using the method get_builtin_subscriber provided by the SubscriberFactory. The built-in DataReader objects can be retrieved by using the operation get_datareader, with the Subscriber and the topic name as parameters.

The QoS of the built-in Subscriber and DataReader objects is given by the following table:
The following tables describe those built-in topics as well as their contents.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_DATA</td>
<td>&lt;unspecified&gt;</td>
</tr>
<tr>
<td>DURABILITY</td>
<td>TRANSIENT</td>
</tr>
<tr>
<td>PRESENTATION</td>
<td>access_scope = TOPIC, coherent_access = FALSE, ordered_access = FALSE</td>
</tr>
<tr>
<td>DEADLINE</td>
<td>Period = infinite</td>
</tr>
<tr>
<td>LATENCY_BUDGET</td>
<td>duration = &lt;unspecified&gt;</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>SHARED</td>
</tr>
<tr>
<td>OWNERSHIP_STRENGTH</td>
<td>&lt;unspecified&gt;</td>
</tr>
<tr>
<td>LIVELINESS</td>
<td>kind = AUTOMATIC, lease_duration = &lt;unspecified&gt;</td>
</tr>
<tr>
<td>TIME_BASED_FILTER</td>
<td>minimum_separation = 0</td>
</tr>
<tr>
<td>PARTITION</td>
<td>&lt;unspecified&gt;</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>kind = RELIABLE</td>
</tr>
<tr>
<td>DESTINATION_ORDER</td>
<td>BY_RECEPTION_TIMESTAMP</td>
</tr>
<tr>
<td>HISTORY</td>
<td>kind = KEEP_LAST, depth = 1</td>
</tr>
<tr>
<td>RESOURCE_LIMITS</td>
<td>All unlimited.</td>
</tr>
</tbody>
</table>

The following tables describe those built-in topics as well as their contents.
### 2.1.6 Interaction Model

Two interaction models are shown here to illustrate the behavior of the DCPS. The first one concerns publication, the second one subscription.

<table>
<thead>
<tr>
<th>Topic/Field</th>
<th>Name Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCPSParticipant</strong> (entry created when a <strong>DomainParticipant</strong> object is created)</td>
<td>key</td>
<td>DCPSKey</td>
</tr>
<tr>
<td></td>
<td>DCPS key to distinguish entries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>user_data</td>
<td>sequence of octets</td>
</tr>
<tr>
<td></td>
<td>Data that can be set when the related DomainParticipant is created (via the USER_DATA <strong>QosPolicy</strong> and that the application may use as it wishes (e.g., to perform some security checking).</td>
<td></td>
</tr>
<tr>
<td><strong>DCPSTopic</strong> (entry created when a <strong>Topic</strong> object is created)</td>
<td>key</td>
<td>DCPSKey</td>
</tr>
<tr>
<td></td>
<td>DCPS key to distinguish entries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>name of the Topic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type_name</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>name of the attached type</td>
<td></td>
</tr>
<tr>
<td><strong>DCPSPublication</strong> (entry created when a <strong>DataWriter</strong> is created in association with its <strong>Publisher</strong>)</td>
<td>topic_name</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>related topic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partition</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>name of the partition (given via the PARTITION <strong>QosPolicy</strong>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>user_data</td>
<td>sequence of octets</td>
</tr>
<tr>
<td></td>
<td>Data that can be set when the related Publisher is created (via the USER_DATA <strong>QosPolicy</strong>) and that the application may use as it wishes (e.g., to perform some security checking).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>key</td>
<td>DCPSKey</td>
</tr>
<tr>
<td></td>
<td>DCPS key to distinguish entries</td>
<td></td>
</tr>
<tr>
<td><strong>DCPSSubscription</strong> (entry created when a <strong>DataReader</strong> is created in association with its <strong>Subscriber</strong>)</td>
<td>topic_name</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>related topic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partitions</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>names of the searched partitions (concatenation of those given via the PARTITION <strong>QosPolicy</strong> objects) separated by ';'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>user_data</td>
<td>sequence of octets</td>
</tr>
<tr>
<td></td>
<td>Data that can be set when the related Publisher is created (via the USER_DATA <strong>QosPolicy</strong>) and that the application may use as it wishes (e.g., to perform some security checking).</td>
<td></td>
</tr>
</tbody>
</table>
It should be noted that these models are not intended to explain how the Service is implemented. In particular, what happens on the right side of the picture (e.g., which components actually send the notifications) should be understood as how it may work rather than how it actually does work (as written inside quotes on the diagrams).

2.1.6.1 Publication View
Figure 2-21  DCPS Interaction Model (Publication)
The first part of Figure 21 shows the **Publisher**'s creation. The second part shows that topics must be created before they are referred to by a **DataWriter**. It should be noted that the topic creation can be requested by a different actor than the one that will use it afterward (in that case, it has to be searched by **TopicFactory::get_topic**).

The third part of Figure 21 shows the **DataWriter**'s creation. Then, a **write** and a **dispose_instance** operation are issued on the **DataWriter**, which immediately informs the **Publisher**. Since the application has not invoked the **suspend_publications** operation on the **Publisher**, the corresponding notifications are propagated according to the current **Publisher**'s policy regarding sending.\(^{34}\)

The last part of Figure 21 shows the same kind of interactions embedded into a pair of **suspend_publications/resume_publications**. It is important to take into account that the corresponding notifications are now delayed until the last **resume_publications**. It should also be noted that even if the diagram shows only one **DataWriter**, several of them could be bracketed within a suspend/resume pair.

### 2.1.6.2 Subscription View

On the subscription side, two diagrams are given. The first one (cf. Figure 22) shows how it works when listeners are used, while the second (cf. Figure 23) shows the use of conditions and wait-sets.

#### 2.1.6.2.1 Notification via Listeners

The first part of Figure 22 shows the **Subscriber**'s and the **DataReader**'s creation by means of the **SubscriberFactory**.

The second part shows the use of a **SubscriberListener**: It must first be created and attached to the **Subscriber** (set_listener). Then when data arrives, it is made available to each related **DataReader**. Then the **SubscriberListener** is triggered (on_data_on_readers). The application must then get the list of affected **DataReader** objects (get_datareaders); then it can read/take the data directly on these objects.

Alternatively, the third part of the diagram shows the use of **DataReaderListener** objects which are first created and attached to the readers. When data is made available on a **DataReader**, the related listener is triggered and data can be read (read/take). It should be noted that, in this case, no link between readers can be retrieved.

**Note**: when the two kinds of listeners are set, the **SubscriberListener** supersedes the **DataReaderListener** ones.

---

\(^{34}\)Usually, this means that the notifications are sent immediately.
Figure 2-22  DCPS Interaction Model (Subscription with Listeners)
2.1.6.2.2 Notifications via Conditions and Wait-Sets

The first part of Figure 22 shows the Subscriber's and the DataReader's creation by means of the SubscriberFactory.

The second part shows the creation of a WaitSet and a ReadCondition, the attachment of the latter to the former, and the call to the WaitSet::wait operation. Note that it is likely that several conditions (ReadCondition, but also StatusCondition) will be created and attached to the same WaitSet.

The third part of the diagram shows the information flow when data is received. Note that the wait returns the list of all the enabled conditions, in an arrival cycle: in case several DataReader objects receive available data, several conditions will be set enabled at the same time and the application will perform several read accordingly.

Note – With conditions and wait-sets, read operations are executed naturally in the user context.
Figure 2-23  DCPS Interaction Model (Subscriptions with Conditions)
2.2 CORBA Platform Specific Model (PSM)

2.2.1 Introduction

The CORBA PIM is provided by means of the IDL that defines the interface an application can use to interact with the Service.

2.2.2 PIM to PSM Mapping Rules

A key concern in the development of the interface is performance. This is a consequence of the application space targeted by the Data Distribution Service (DDS).

The PIM to PSM mapping maps the UML interfaces and classes interfaces into IDL interfaces. Plain data types are mapped into structures.

IDL interfaces do not support overloading. The cases where a base class or interface has an abstract operation that must be redefined by an specialized class or interface has been mapped into a base IDL interface where the abstract operation appears inside comments. This serves simply as a reminder of the fact that all specializations must implement the operation.

Enumerations have been mapped into either IDL ‘enum’ or hand-picked IDL ‘long’ values that correspond to increasing powers of 2 (that is 0x01, 0x02, 0x04, etc.). The latter choice has been made when it was required to refer to multiple values as part of a function parameter. This allowed the use of a ‘long’ as a mask to indicate a set of enumerated values. This selection affected only the PIM ‘status kind’ values, namely: `StatusKind`, `SampleStateKind`, and `LifecycleStateKind`.

Collection parameters have been mapped into IDL sequences. The only exception applies to the case where the collection elements are hand-picked IDL ‘long’. In this case the collection is mapped into an IDL ‘long’ interpreted as a mask.

Each `QosPolicy` has been mapped as an IDL struct. The collection of policies suitable for each Entity has been modelled as another IDL struct that contains attributes corresponding to the policies that apply to this `Entity`. This approach has several advantages. First, it provides compile-time checking of the suitability of setting a specific `QosPolicy` on a particular `Entity`. A second advantage is that it does not require the use of the type “any” which increases code size and is not natural to use in “C.” Other approaches were less attractive. IDL interfaces are not suitable because a collection of `QosPolicy` appears as an argument to several operations and needs to be passed “by value.” IDL ‘valuetype’ was considered but rejected because it is not universally supported and also forces each attribute to be accessed via an operation.

Error-return values have been mapped to plain return codes of the corresponding functions. The reason is that DCPS targets “C” as one of the key deployment languages and return codes are more natural to use in “C.”
2.2.3 DCPS PSM: IDL

```c
#define DOMAINID_TYPE_NATIVE      long
#define HANDLE_TYPE_NATIVE        long
#define HANDLE_NIL_NATIVE         0

module DCPS {
    typedef DOMAINID_TYPE_NATIVE  DomainId_t;
typedef HANDLE_TYPE_NATIVE     InstanceHandle_t;

typedef long      ReturnCode_t;
typedef long      QosPolicyId_t;
typedef sequence<string> StringSeq;

struct Duration_t {
    long   sec;
    unsigned long nanosec;
};

struct Time_t {
    long   sec;
    unsigned long nanosec;
};

// Pre-defined values
const InstanceHandle_t HANDLE_NIL = HANDLE_NIL_NATIVE;

// Return codes
const ReturnCode_t RETCODE_OK                    = 0;
const ReturnCode_t RETCODE_ERROR                = 1;
const ReturnCode_t RETCODE_UNSUPPORTED          = 2;
const ReturnCode_t RETCODE_BAD_PARAMETER        = 3;
const ReturnCode_t RETCODE_PRECONDITION_NOT_MET  = 4;
const ReturnCode_t RETCODE_OUT_OF_RESOURCES     = 5;
const ReturnCode_t RETCODE_NOT_ENABLED          = 6;
const ReturnCode_t RETCODE_IMMUTABLE_POLICY      = 7;
const ReturnCode_t RETCODE_INCONSISTENT_POLICY   = 8;

// Status to support listeners and conditions
const StatusKind INCONSISTENT_TOPIC_STATUS      = 0x0001 << 0;
const StatusKind OFFERED_DEADLINE_MISSED_STATUS = 0x0001 << 1;
const StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2;
const StatusKind OFFERED_INSTANCE_DEADLINE_MISSED_STATUS = 0x0001 << 3;
const StatusKind REQUESTED_INSTANCE_DEADLINE_MISSED_STATUS = 0x0001 << 4;
```
const StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS = 0x0001 << 5;
const StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS = 0x0001 << 6;
const StatusKind SAMPLE_LOST_STATUS = 0x0001 << 7;
const StatusKind SAMPLE_REJECTED_STATUS = 0x0001 << 8;
const StatusKind DATA_ON_READERS_STATUS = 0x0001 << 9;
const StatusKind DATA_AVAILABLE_STATUS = 0x0001 << 10;

struct InconsistentTopicStatus {
    long total_count;
    long total_count_change;
};

struct SampleLostStatus {
    long total_count;
    long total_count_change;
};

enum SampleRejectedStatusKind {
    REJECTED_BY_INSTANCE_LIMIT,
    REJECTED_BY_TOPIC_LIMIT
};

struct SampleRejectedStatus {
    long total_count;
    long total_count_change;
    SampleRejectedStatusKind last_reason;
    InstanceHandle_t last_instance_handle;
};

struct LivelinessLostStatus {
    long total_count;
    long total_count_change;
};

struct LivelinessChangedStatus {
    long active_count;
    long inactive_count;
    long active_count_change;
    long inactive_count_change;
};

struct OfferedDeadlineMissedStatus {
    long total_count;
    long total_count_change;
    InstanceHandle_t last_instance_handle;
};

struct RequestedDeadlineMissedStatus {
    long total_count;
    long total_count_change;
    InstanceHandle_t last_instance_handle;
};

struct QosPolicyCount {

QosPolicyId_t policy_id;
long count;

};

struct OfferedIncompatibleQosStatus {
    long total_count;
    long total_count_change;
    QosPolicyId_t last_policy_id;
    sequence<QosPolicyCount> policies;
};

struct RequestedIncompatibleQosStatus {
    long total_count;
    long total_count_change;
    QosPolicyId_t last_policy_id;
    sequence<QosPolicyCount> policies;
};

// Listeners

interface Listener;
interface Entity;
interface Topic;
interface ContentFilteredTopic;
interface MultiTopic;
interface DataWriter;
interface DataReader;
interface Subscriber;
interface Publisher;

typedef sequence<Topic> TopicSeq;
typedef sequence<DataReader> DataReaderSeq;

interface Listener {
    interface TopicListener : Listener {
        void on_inconsistent_topic(in Topic topic,
            in InconsistentTopicStatus status);
    }
}

interface DataWriterListener : Listener {
    void on_offered_deadline_missed(in DataWriter writer,
        in OfferedDeadlineMissedStatus status);
    void on_offered_incompatible_qos(in DataWriter writer,
        in OfferedIncompatibleQosStatus status);
    void on_liveliness_lost(in DataWriter writer,
        in LivelinessLostStatus status);
};

interface PublisherListener : DataWriterListener {
};

interface DataReaderListener : Listener {
}
void onRequestedDeadlineMissed(in DataReader reader,
in RequestedDeadlineMissedStatus status);
void onRequestedIncompatibleQos(in DataReader reader,
in RequestedIncompatibleQosStatus status);
void onSampleRejected(in DataReader reader,
in SampleRejectedStatus status);
void onLivelinessChanged(in DataReader reader,
in LivelinessChangedStatus status);
void onDataAvailable(in DataReader reader);
};

interface SubscriberListener : DataReaderListener {
  void onSampleLost(in Subscriber subs,
in SampleLostStatus status);
  void onDataOnReaders(in Subscriber subs);
};

interface DomainParticipantListener : TopicListener,
PublisherListener,
SubscriberListener {
};

// ----------------------------------------------------------------------
// Conditions
// ----------------------------------------------------------------------

interface Condition {
  boolean get_trigger_value();
};

typedef sequence<Condition> ConditionSeq;

interface WaitSet {
  ReturnCode_t wait(out ConditionSeq active_conditions,
in Duration_t timeout);
  ReturnCode_t attach_condition(in Condition cond);
  ReturnCode_t detach_condition(in Condition cond);
};

interface GuardCondition : Condition {
  void set_trigger_value(in boolean value);
};

interface StatusCondition : Condition {
  readonly attribute StatusKindMask enabled_statuses;
  ReturnCode_t set_enabled_statuses(in StatusKindMask mask);
};

// Instance states to support reads
typedef unsigned long SampleStateKind;
typedef sequence <SampleStateKind> SampleStateSeq;
const SampleStateKind READSAMPLESTATE = 0x0001 << 0;
const SampleStateKind NOTREADSAMPLESTATE = 0x0001 << 1;
// This is a bit-mask SampleStateKind
typedef unsigned long SampleStateMask;
const SampleStateMask ANY_SAMPLE_STATE = 0xffff;

// Sample states to support reads
typedef unsigned long LifecycleStateKind;
typedef sequence<LifecycleStateKind> LifecycleStateSeq;
const LifecycleStateKind NEW_LIFECYCLE_STATE = 0x0001 << 0;
const LifecycleStateKind MODIFIED_LIFECYCLE_STATE = 0x0001 << 1;
const LifecycleStateKind DISPOSED_LIFECYCLE_STATE = 0x0001 << 2;
const LifecycleStateKind NO_WRITERS_LIFECYCLE_STATE = 0x0001 << 3;

// This is a bit-mask LifecycleStateKind
typedef unsigned long LifecycleStateMask;
const LifecycleStateMask ANY_LIFECYCLE_STATE = 0xffff;

interface ReadCondition : Condition {
    readonly attribute LifecycleStateMask lifecycle_state_mask;
    readonly attribute SampleStateMask sample_state_mask;
};

interface QueryCondition : ReadCondition {
    readonly attribute string query_expression;
    attribute StringSeq query_arguments;
};

// Qos
const string USERDATA_QOS_POLICY_NAME = "UserData";
const string DURABILITY_QOS_POLICY_NAME = "Durability";
const string PRESENTATION_QOS_POLICY_NAME = "Presentation";
const string DEADLINE_QOS_POLICY_NAME = "Deadline";
const string LATENCYBUDGET_QOS_POLICY_NAME = "LatencyBudget";
const string OWNERSHIP_QOS_POLICY_NAME = "Ownership";
const string OWNERSHIPSTRENGTH_QOS_POLICY_NAME = "OwnershipStrength";
const string LIVELINESS_QOS_POLICY_NAME = "Liveliness";
const string TIMEBASEDFILTER_QOS_POLICY_NAME = "TimeBasedFilter";
const string PARTITION_QOS_POLICY_NAME = "Partition";
const string RELIABILITY_QOS_POLICY_NAME = "Reliability";
const string DESTINATIONORDER_QOS_POLICY_NAME = "DestinationOrder";
const string HISTORY_QOS_POLICY_NAME = "History";
const string RESOURCELIMITS_QOS_POLICY_NAME = "ResourceLimits";

const QosPolicyId_t USERDATA_QOS_POLICY_ID = 1;
const QosPolicyId_t DURABILITY_QOS_POLICY_ID = 2;
const QosPolicyId_t PRESENTATION_QOS_POLICY_ID = 3;
const QosPolicyId_t DEADLINE_QOS_POLICY_ID = 4;
const QosPolicyId_t LATENCYBUDGET_QOS_POLICY_ID = 5;
const QosPolicyId_t OWNERSHIP_QOS_POLICY_ID = 6;
const QosPolicyId_t OWNERSHIPSTRENGTH_QOS_POLICY_ID = 7;
const QosPolicyId_t LIVELINESS_QOS_POLICY_ID = 8;
const QosPolicyId_t TIMEBASEDFILTER_QOS_POLICY_ID = 9;
const QosPolicyId_t PARTITION_QOS_POLICY_ID = 10;
const QosPolicyId_t RELIABILITY_QOS_POLICY_ID = 11;
const QosPolicyId_t DESTINATIONORDER_QOS_POLICY_ID = 12;
const QosPolicyId_t HISTORY_QOS_POLICY_ID = 13;
const QosPolicyId_t RESOURCESLIMITS_QOS_POLICY_ID = 14;

struct UserDataQosPolicy {
    sequence<octet> data;
};

enum DurabilityQosPolicyKind {
    VOLATILE_DURABILITY_QOS,
    TRANSIENT_DURABILITY_QOS,
    PERSISTENT_DURABILITY_QOS
};
struct DurabilityQosPolicy {
    DurabilityQosPolicyKind kind;
};

enum PresentationQosPolicyAccessScopeKind {
    INSTANCE_PRESENTATION_QOS,
    TOPIC_PRESENTATION_QOS,
    GROUP_PRESENTATION_QOS
};
struct PresentationQosPolicy {
    PresentationQosPolicyAccessScopeKind access_scope;
    boolean coherent_access;
    boolean ordered_access;
};

struct DeadlineQosPolicy {
    Duration_t period;
};

struct LatencyBudgetQosPolicy {
    Duration_t duration;
};

enum OwnershipQosPolicyKind {
    SHARED_OWNERSHIP_QOS,
    EXCLUSIVE_OWNERSHIP_QOS
};
struct OwnershipQosPolicy {
    OwnershipQosPolicyKind kind;
};

struct OwnershipStrengthQosPolicy {
    long value;
};

enum LivelinessQosPolicyKind {
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS
};
struct LivelinessQosPolicy {
    LivelinessQosPolicyKind kind;
    Duration_t lease_duration;
};

struct TimeBasedFilterQosPolicy {
    Duration_t minimum_separation;
};

struct PartitionQosPolicy {
    StringSeq name;
};

enum ReliabilityQosPolicyKind {
    BEST_EFFORT_RELIABILITY_QOS,
    RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy {
    ReliabilityQosPolicyKind kind;
};

enum DestinationOrderQosPolicyKind {
    BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
    BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};

struct DestinationOrderQosPolicy {
    DestinationOrderQosPolicyKind kind;
};

enum HistoryQosPolicyKind {
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};

struct HistoryQosPolicy {
    HistoryQosPolicyKind kind;
    long depth;
};

struct ResourceLimitsQosPolicy {
    long max_samples;
    long max_instances;
    long max_samples_per_instance;
};

struct DomainParticipantQos {
    UserDataQosPolicy user_data;
};

struct TopicQos {
    DurabilityQosPolicy durability;
    DeadlineQosPolicy deadline;
    LatencyBudgetQosPolicy delay_laxity;
    LivelinessQosPolicy liveliness;
}
struct DataWriterQos {
    DurabilityQosPolicy          durability;
    DeadlineQosPolicy            deadline;
    LatencyBudgetQosPolicy       delay_laxity;
    LivelinessQosPolicy          liveliness;
    ReliabilityQosPolicy         reliability;
    DestinationOrderQosPolicy    destination_order;
    HistoryQosPolicy             history;
    ResourceLimitsQosPolicy      resource_limits;
    OwnershipQosPolicy           ownership;
    UserDataQosPolicy            user_data;
    OwnershipStrengthQosPolicy   ownership_strength;
};

struct PublisherQos {
    UserDataQosPolicy            user_data;
    PresentationQosPolicy        presentation;
    PartitionQosPolicy           partition;
};

struct DataReaderQos {
    DurabilityQosPolicy          durability;
    DeadlineQosPolicy            deadline;
    LatencyBudgetQosPolicy       delay_laxity;
    LivelinessQosPolicy          liveliness;
    ReliabilityQosPolicy         reliability;
    DestinationOrderQosPolicy    destination_order;
    HistoryQosPolicy             history;
    ResourceLimitsQosPolicy      resource_limits;
    UserDataQosPolicy            user_data;
    TimeBasedFilterQosPolicy     time_based_filter;
};

struct SubscriberQos {
    UserDataQosPolicy            user_data;
    PresentationQosPolicy        presentation;
    PartitionQosPolicy           partition;
};

// ----------------------------------------------------------------------
interface Entity {
//    ReturnCode_t set_qos(in DataWriterQos qos);
//    void get_qos(inout DataWriterQos qos);
//    ReturnCode_t set_listener(in Listener l, in StatusKindMask mask);
//    Listener get_listener();
ReturnCode_t enable();
void assert_liveliness();

StatusCondition create_statuscondition(in StatusKindMask mask);
ReturnCode_t delete_statuscondition(in StatusCondition the_condition);

StatusKindMask get_status_changes();
};

// ----------------------------------------------------------------------
interface DomainParticipant : Entity {
    // Factory interfaces
    Publisher create_publisher(in PublisherQos qos,
in PublisherListener listener);
    ReturnCode_t delete_publisher(in Publisher p);

    Subscriber create_subscriber(in SubscriberQos qos,
in SubscriberListener listener);
    ReturnCode_t delete_subscriber(in Subscriber s);
    Subscriber get_builtin_subscriber();

    Topic create_topic(in string topic_name, in string type_name,
in TopicQos qos, in TopicListener listener);
    ReturnCode_t delete_topic(in Topic topic);
    Topic lookup_topic(in string topic_name, in Duration_t timeout);

    ContentFilteredTopic create_contentfilteredtopic(in string name,
in Topic related_topic, in string filter_expression,
in StringSeq filter_parameters);
    ReturnCode_t delete_contentfiltered(in ContentFilteredTopic a_contentfilteredtopic);

    MultiTopic create_multitopic(in string name, in string type_name,
in string subscription_expression, in StringSeq expression_parameters);
    ReturnCode_t delete_multitopic(in MultiTopic a_multitopic);

    ReturnCode_t set_qos(in DomainParticipantQos qos);
    void get_qos(inout DomainParticipantQos qos);

    ReturnCode_t ignore_participant(in InstanceHandle_t handle);
    ReturnCode_t ignore_topic(in InstanceHandle_t handle);
    ReturnCode_t ignore_publication(in InstanceHandle_t handle);
    ReturnCode_t ignore_subscription(in InstanceHandle_t handle);

    readonly attribute DomainId_t domainId;
};
DomainParticipant create_participant(in DomainId_t domainId,
    in DomainParticipantQos qos,
    in DomainParticipantListener listener);
    ReturnCode_t delete_participant(in DomainParticipant a_participant);
};

interface DataType {
    //    ReturnCode_t register_type(in DomainParticipant domain, in string type_name);
};

// ---------------------------------------------------------------------------
interface TopicDescription {
    readonly attribute string type_name;
};

interface Topic : Entity, TopicDescription {
    readonly attribute string name;

    // Access the status
    readonly attribute InconsistentTopicStatus inconsistent_topic_status;
};

interface ContentFilteredTopic : TopicDescription {
    readonly attribute string filter_expression;
    attribute StringSeq expression_parameters;
};

interface MultiTopic : TopicDescription {
    readonly attribute string topic_expression;
    attribute StringSeq expression_parameters;
};

// ---------------------------------------------------------------------------
interface Publisher : Entity {
    DataWriter create_datawriter(in Topic t, in DataWriterQos qos,
        in DataWriterListener listener);
    ReturnCode_t delete_datawriter(in DataWriter a_datawriter);
    DataWriter lookup_datawriter(in string topic_name);

    ReturnCode_t set_qos(in PublisherQos qos);
    void get_qos(inout PublisherQos qos);

    void suspend_publications();
    void resume_publications();

    ReturnCode_t begin_coherent_changes();
    ReturnCode_t end_coherent_changes();
};

interface DataWriter : Entity {
    //    InstanceHandle_t register_instance(in Data instance_data);
    //    ReturnCode_t unregister_instance(in Data instance_data,
    //        in InstanceHandle_t handle);
    //    ReturnCode_t write(in Data instance_data, in InstanceHandle_t handle);
    //    ReturnCode_t write_w_timestamp(in Data instance_data,
// in InstanceHandle_t handle,
// in Time_t source_timestamp);
// ReturnCode_t dispose(in Data instance_data,
// in InstanceHandle_t instance_handle);
// ReturnCode_t dispose_w_timestamp(in Data instance_data,
// in InstanceHandle_t instance_handle,
// in Time_t source_timestamp);
// ReturnCode_t get_key(inout Data key_holder, in InstanceHandle_t handle);

ReturnCode_t set_qos(in DataWriterQos qos);
void get_qos(inout DataWriterQos qos);

// Access the status
readonly attribute LivelinessLostStatus liveliness_lost_status;
readonly attribute OfferedDeadlineMissedStatus offered_deadline_missed_status;
readonly attribute OfferedIncompatibleQosStatus offered_incompatible_qos_status;
}

// ----------------------------------------------------------------------
interface Subscriber : Entity {
    DataReader create_datareader(in TopicDescription topic,
        in DataReaderQos qos,
        in DataReaderListener listener);
    ReturnCode_t delete_datareader(in DataReader a_datareader);
    DataReader lookup_datareader(in string topic_name);
    ReturnCode_t get_datareaders(out DataReaderSeq readers,
        in LifecycleStateMask l_state,
        in SampleStateMask s_state);
    void notify_datareaders(in LifecycleStateMask l_state,
        in SampleStateMask s_state);
    ReturnCode_t set_qos(in SubscriberQos qos);
    void get_qos(inout SubscriberQos qos);
    ReturnCode_t begin_access();
    ReturnCode_t end_access();
    readonly attribute SampleLostStatus sample_lost_status;
}

interface DataReader : Entity {
    // ReturnCode_t read(out DataSeq received_data,
    // out SampleInfoSeq info_seq,
    // in SampleStateMask s_mask,
    // in LifecycleStateMask l_mask);
    // ReturnCode_t take(out DataSeq received_data,
    // out SampleInfoSeq info_seq,
    // in SampleStateMask s_mask,
    // in LifecycleStateMask l_mask);
    // ReturnCode_t read_w_condition(out DataSeq received_data,
    // out SampleInfo info_seq,
    // in ReadCondition condition);
// ReturnCode_t take_w_condition(out DataSeq received_data,
//       out SampleInfo info_seq,
//       in ReadCondition condition);

// ReturnCode_t get_key(inout Data key_holder, in InstanceHandle_t handle);

ReadCondition create_readcondition(in SampleStateMask mask,
       in LifecycleStateMask l_mask);

QueryCondition
create_querycondition(in SampleStateMask mask,
       in LifecycleStateMask l_mask,
       in string query,
       in StringSeq query_parameters);

ReturnCode_t delete_readcondition(in ReadCondition condition);

ReturnCode_t set_qos(in DataReaderQos qos);
void get_qos(inout DataReaderQos qos);

readonly attribute SampleRejectedStatus sample_rejected_status;
readonly attribute LivelinessChangedStatus liveliness_changed_status;
readonly attribute RequestedDeadlineMissedStatus requested_deadline_missed_status;
readonly attribute RequestedIncompatibleQosStatus requested_incompatible_qos_status;
};

struct SampleInfo {
    SampleStateKind  sample_state;
    LifecycleStateKind lifecycle_state;
    Time_t source_timestamp;
    InstanceHandle_t instance_handle;
};

typedef sequence<SampleInfo> SampleInfoSeq;

// Implied IDL for type "Foo"
// Example user defined structure
struct Foo {
    long dummy;
};

typedef sequence<Foo> FooSeq;

#include "dds_dcps.idl"

typedef sequence<Foo> FooSeq;

interface FooDataType : DCPS::DataType {
    DCPS::ReturnCode_t register_type(in DCPS::DomainParticipant participant,
       in string type_name);
};

interface FooDataWriter : DCPS::DataWriter {
DCPS::InstanceHandle_t register_instance(in Foo instance_data);

DCPS::ReturnCode_t unregister_instance(in Foo instance_data,
            in DCPS::InstanceHandle_t handle);

DCPS::ReturnCode_t write(in Foo instance_data,
            in DCPS::InstanceHandle_t handle);

DCPS::ReturnCode_t write_w_timestamp(in Foo instance_data,
            in DCPS::InstanceHandle_t handle,
            in DCPS::Time_t source_timestamp);

DCPS::ReturnCode_t dispose(in Foo instance_data,
            in DCPS::InstanceHandle_t handle);

DCPS::ReturnCode_t dispose_w_timestamp(in Foo instance_data,
            in DCPS::InstanceHandle_t handle,
            in DCPS::Time_t source_timestamp);

DCPS::ReturnCode_t get_key(inout Foo key_holder,
            in DCPS::InstanceHandle_t handle);

});

interface FooDataReader : DCPS::DataReader {
            DCPS::ReturnCode_t read(out FooSeq received_data,
                out DCPS::SampleInfoSeq info_seq,
                in  DCPS::SampleStateMask s_mask,
                in  DCPS::LifecycleStateMask l_mask);

            DCPS::ReturnCode_t take(out FooSeq received_data,
                out DCPS::SampleInfoSeq info_seq,
                in  DCPS::SampleStateMask s_mask,
                in  DCPS::LifecycleStateMask l_mask);

            DCPS::ReturnCode_t read_w_condition(out FooSeq received_data,
                out DCPS::SampleInfoSeq info_seq,
                in  DCPS::ReadCondition condition);

            DCPS::ReturnCode_t take_w_condition(out FooSeq received_data,
                out DCPS::SampleInfoSeq info_seq,
                in  DCPS::ReadCondition condition);

            DCPS::ReturnCode_t get_key(inout Foo key_holder,
                in DCPS::InstanceHandle_t handle);

});
Data Local Reconstruction Layer
(DLRL)

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“PIM”</td>
<td>3-1</td>
</tr>
<tr>
<td>“CORBA PSM”</td>
<td>3-40</td>
</tr>
</tbody>
</table>

3.1 PIM

DLRL stands for Data Local Reconstruction Layer. It is an optional layer that may be built on top of the DCPS layer.

3.1.1 Overview and Design Rationale

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

As for DCPS, typed interfaces\(^1\) have been selected, for the same reasons of ease of use and potential performance.

---

1. In the sense, interfaces whose type depend on application-defined types.
As far as possible, DLRL is designed to allow the application developer to use the underlying DCPS features. However, this may conflict with the main purpose of this layer, which is ease of use and seamless integration into the application. Therefore, some DCPS features may only be used through DCPS and are not accessible from the DLRL.

### 3.1.2 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, etc.;
  - functional mapping i.e., relations between the DLRL functions (mainly access to the DLRL objects) and the DCPS functions (write/publish/etc.).

### 3.1.3 What Can Be Modeled with DLRL

#### 3.1.3.1 DLRL objects

DLRL allows an application to describe objects with:

- methods;
- attributes which 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\(^2\) and can be either mono-valued or multi-valued:

- Mono-valued attributes are of a simple type:
  - basic-type (long, short, char, string, etc.);
  - enumeration-type;
  - simple structure\(^3\) reference to a DLRL object.

Type enforcement is as follows:

- strict type equality for simple types;
- compatibility 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) attributes can be either:
  - a list (ordered with index);
  - a map (access by key).

Items in the collection must be homogeneously-typed.

Type enforcement is as follows:

- same type of collection (list and list, or map and map);
- strict type equality for simple types;
- compatibility 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 \textit{oid} (object ID) part of any DLRL object.

### 3.1.3.2 Relations among DLRL objects

Relations between DLRL objects are of two kinds:

- inheritance that organize the DLRL classes
- associations that organize the DLRL instances

#### 3.1.3.2.1 Inheritance

Single inheritance is allowed between DLRL objects.

---

2. 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.

3. I.e. structures that can be mapped inside one DCPS data.
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.

### 3.1.3.2.2 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)

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.

### 3.1.3.3 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.

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.
Instances of **BasicType** are:
- long
- short
- char
- octet
- real

Figure 3-1  DLRL Metamodel
3.1.4 Structural Mapping

3.1.4.1 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)\(^4\).

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.

3.1.4.2 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.

The mapping rules are as follows:

---

\(^4\) This is needed to efficiently manage inheritance. Therefore extending it to any attribute is not costly.
3.1.4.2.1 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).

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)\(^5\), 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.

3.1.4.2.2 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\(^6\) (\textit{oid}).

The combination of these two pieces of information is called \textit{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).

3.1.4.2.3 Mapping of Attributes and Relations

Mono-valued attributes and relations are mapped to one (or several) cell(s)\(^7\) in a single row whose key is the means to unambiguously reference the DLRL object (i.e., its \textit{oid} or its full \textit{oid}, depending on the \textit{owner} class characteristics as indicated in the previous section):

- simple basic attributes -> one cell of corresponding DCPS type;
- enumeration -> one cell of type integer\(^8\) (default behavior) or string;

---

5. Excluding, of course, the abstract ObjectRoot (otherwise all the objects will be located in a single table).

6. 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.

7. Depending of the type of the value.
• 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).

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 which 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 if the collection basis is a map (to hold the access key of the item in the map).

3.1.4.3 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.

8. 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.
9. String-keyed maps are desired for their openness; however, integer-keyed maps are more suitable when access performance is desired.
• 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:
    • 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.
3.1.4.4 Metamodel with Mapping Information

Figure 3-2 represents the DLRL metamodel with the information that is needed to indicate the structural mapping.
The three constructs that need to be added information related to the structural mapping are Class, Attribute and Relation.

### 3.1.4.4.1 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\(^{10}\);
- **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;

### 3.1.4.4.2 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::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);

### 3.1.4.4.3 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)\(^{11}\).

---

10. It may have attributes in other topics as well.
11. 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).
3.1.4.4 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);

- **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;

3.1.4.5 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_fields** 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;

3.1.4.5 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 which 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 3-3.
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 CORBA PSM in Section 3.1.2, “DLRL Description,” on page 3-2.

### 3.1.5 Operational Mapping

#### 3.1.5.1 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 **Relation** objects. 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.

### 3.1.5.2 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.

### 3.1.5.3 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, etc.). 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.

### 3.1.6 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).

### 3.1.6.1 DLRL Requested Functions

#### 3.1.6.1.1 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.

### 3.1.6.1.2 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.

#### 3.1.6.1.2.1 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\(^\text{12}\), 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?).
- 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 (Null-Pointer).

#### 3.1.6.1.2.2 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.

\(^{12}\) That means that no subscription will be made "on the fly" to reach an object which is an instance of a class for which no subscription has been made.
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 (cf. CacheAccess objects for more details).

### 3.1.6.1.2.3 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.

### 3.1.6.1.3 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.

### 3.1.6.2 DLRL Entities

Figure 3-4 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 3.1.6.6, “Generated Classes,” on page 3-40.
Figure 3-4 DLRL entities
The DLRL entities are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheFactory</td>
<td>Class whose unique instance allows the creation of Cache objects.</td>
</tr>
</tbody>
</table>
| Cache          | Class whose instance represents a set of objects that are locally available. Several Cache objects may be created. However, 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 which encapsulates the access to a set of objects. It offers methods to load, refresh, write objects attached to it; a Cache has by construction one CacheAccess (by inheritance); in addition, other CacheAccess objects can be created 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.                    |
| 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.       |
| ObjectFilter   | Class whose instances act as filter for Selection objects. When a Selection is created, it must be given an ObjectFilter.                       |
| ObjectQuery    | Specialization of the above that perform 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.                                                                                |
| ObjectLink     | Class to represent a link to another object.                                                                                                 |
| Reference      | Class to represent a reference (i.e., a typed link) to another object.                                                                          |
| 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.                                                                            |
| 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.                                                   |
| Relation       | Abstract root for all the relations between objects.                                                                                         |
3.1.6.3 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.

3.1.6.3.1 CacheFactory

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

<table>
<thead>
<tr>
<th>RefRelation</th>
<th>Abstract root for all the classes to represent a to-1 relation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListRelation</td>
<td>Abstract root for all the classes to represent a to-n relation based on a list.</td>
</tr>
<tr>
<td>StrMapRelation</td>
<td>Abstract root for all the classes to represent a to-n relation based on a map with string key management.</td>
</tr>
<tr>
<td>IntMapRelation</td>
<td>Abstract root for all the classes to represent a to-n relation based on a map with integer key management.</td>
</tr>
</tbody>
</table>

This class offers to unique method to create **Cache** objects:

- **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 the concerned **domain**; 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 **domain**.
3.1.6.3.2 CacheAccess

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

<table>
<thead>
<tr>
<th>CacheAccess</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>cache_usage</td>
</tr>
<tr>
<td>owner</td>
</tr>
<tr>
<td>refs</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>refresh</td>
</tr>
<tr>
<td>write</td>
</tr>
<tr>
<td>purge</td>
</tr>
<tr>
<td>deref</td>
</tr>
<tr>
<td>ref</td>
</tr>
</tbody>
</table>

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

The attribute 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 must be compatible with the value of the owning Cache (cf. Cache::create_access).

Once the CacheAccess is created for a given purpose, copies of DLRL objects can be attached to it (cf. ObjectRoot::clone method), by means of references (refs) and then:

- the attached objects can be refreshed (refresh);
- the copies can be consulted; navigation is performed only into the set of objects attached to the CacheAccess (in other words, if a relation points to an object that has not been attached to the CacheAccess, navigating through that relation will raise an error – Null-Pointer);
- if the 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 (write);
- the copies can be detached from the CacheAccess (purge);
- a method allows transformation of an ObjectLink in the ObjectRoot which is valid for this CacheAccess (deref).

Cf. Section 3.1.6.5 for a description of typical uses of cache accesses.
3.1.6.3.3 Cache

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

<table>
<thead>
<tr>
<th>Cache : CacheAccess</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>pubsub_state</td>
</tr>
<tr>
<td>updates_enabled</td>
</tr>
<tr>
<td>sub_accesses</td>
</tr>
<tr>
<td>homes</td>
</tr>
<tr>
<td>listener</td>
</tr>
<tr>
<td>publisher</td>
</tr>
<tr>
<td>subscriber</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>register_home</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>find_home</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>register_all_for_pubsub</td>
</tr>
<tr>
<td>enable_all_for_pubsub</td>
</tr>
<tr>
<td>attach_listener</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>detach_listener</td>
</tr>
<tr>
<td>enable_updates</td>
</tr>
<tr>
<td>disable_updates</td>
</tr>
<tr>
<td>create_access</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>delete_access</td>
</tr>
<tr>
<td>load</td>
</tr>
</tbody>
</table>

The public attributes give:

- the usage mode of the cache (WRITE_ONLY—no subscription, READ_ONLY—no publication, or READ_WRITE—both modes); this mode applies to all objects in the cache and has to be given at creation time (cache_usage – inherited);
- the state of the cache with respect to the underlying Pub/Sub infrastructure (pubsub_state);
- the state of the cache with respect to incoming updates (updates_enabled); this state is modifiable by the applications (cf. enable_updates, disable_updates) in order to support applications that are both publishing and subscribing.

It offers methods to:

- register an ObjectHome (register_home);
• retrieve an already registered ObjectHome based on its name (find_home);
• 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;
• enable the derived Pub/Sub infrastructure (enable_all_for_pubsub); QoS setting can be performed between those two operations;
• attach/detach a CacheListener (attach_listener, detach_listener);
• enable/disable incoming updates (enable_updates, 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 (cf. 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;
• 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) – 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 sub-accesses that allow writing;
• delete sub-accesses (delete_access);
• request all the known ObjectHome to load all their instances – i.e., instantiate the related DLRL objects (load).

13: That for a sub-access are equivalent.
3.1.6.3.4 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.

- **start_updates** to indicate that updates are following. Actual modifications in the cache will be performed only when exiting this method (assuming that `updates_enabled` is true);
- **end_updates** that indicates that no more update is foreseen (either because no more update has been received – `interrupted` = FALSE, or because the updates have been disabled for that `Cache` – `interrupted` = TRUE);

It provides the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on_begin_updates</code></td>
<td>Indicates that updates are following. Actual modifications in the cache will be performed only when exiting this method (assuming that <code>updates_enabled</code> is true);</td>
</tr>
<tr>
<td><code>on_end_updates</code></td>
<td>Indicates that no more update is foreseen (either because no more update has been received – <code>interrupted</code> = FALSE, or because the updates have been disabled for that <code>Cache</code> – <code>interrupted</code> = TRUE);</td>
</tr>
</tbody>
</table>

In between, the updates are reported on home or selection listeners. Section 3.1.6.4 describes which notifications are performed and in what order.

3.1.6.3.5 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*.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_name</td>
<td>String</td>
</tr>
<tr>
<td>filter</td>
<td>String</td>
</tr>
<tr>
<td>notification_scope</td>
<td>ObjectScope</td>
</tr>
<tr>
<td>extent</td>
<td>ObjectRoot []</td>
</tr>
<tr>
<td>selections</td>
<td>HomeSelection []</td>
</tr>
<tr>
<td>listener</td>
<td>ObjectListener</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_topic_name</td>
<td>String</td>
</tr>
<tr>
<td>attribute_name</td>
<td>String</td>
</tr>
</tbody>
</table>
The public attributes give:

- the public name of the application-defined class \((\text{class\_name})\);
- a filter that is used to filter incoming objects; it only concerns subscribing applications; only the incoming objects that pass the filter will be created in the \text{Cache} and by that \text{ObjectHome}; this 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; cf. Appendix A for its syntax;
- the scope to be considered to determine if an object has been modified and should then lead to listeners activation \((\text{notification\_scope})\). An object can be considered as modified only if its own content has changed, or if its contents as well as the contents of its component objects has changed or if its contents as well as the contents of its related objects has changed;
- list of all the objects of that class \((\text{extent})\).

It offers methods to:

- set the filter for that \text{ObjectHome} \((\text{set\_filter})\); as a filter is intended to be mapped on the underlying infrastructure it can be set only before the \text{ObjectHome} is registered (cf. \text{Cache::register\_home});
- attach/detach an \text{ObjectListener} \((\text{attach\_listener}, \text{detach\_listener})\);
- create a Selection \((\text{create\_selection})\); the filter parameter specifies the \text{ObjectFilter} to be attached to the Selection and the auto\_refresh parameter specifies if the Selection has to be refreshed automatically or only on demand (cf. Selection); attached \text{ObjectFilter} belongs to the Selection that itself belongs to its creating ObjectHome;
- delete a Selection \((\text{delete\_selection})\); this operation deletes the Selection and its attached \text{ObjectFilter};
• create a new DLRL object \textit{(create\_object)}; this operation takes as parameter the \textit{CacheAccess} concerned by the creation;

• retrieve a DLRL object based on its \textit{oid} \textit{(find\_object)}.

3.1.6.3.6 \textbf{ObjectListener}

This interface is an abstract root, from which a typed interface will be derived for each application type. This typed interface (named \textit{FooObjectListener}, if the application class is named \textit{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 \textit{FooHome}.

<table>
<thead>
<tr>
<th>\textbf{ObjectListener}</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
</tr>
<tr>
<td>on_created_object</td>
</tr>
<tr>
<td>object</td>
</tr>
<tr>
<td>on_modified_object</td>
</tr>
<tr>
<td>object</td>
</tr>
<tr>
<td>on_deleted_object</td>
</tr>
<tr>
<td>object</td>
</tr>
</tbody>
</table>

It is defined with three methods:

• \textit{on\_object\_created}, which is called when a new object appears in the \textit{Cache};

• \textit{on\_object\_deleted}, which is called when an object has been deleted by another participant;

• \textit{on\_object\_modified}, which is called when the contents of an object changes.

Each of these methods must return a boolean. TRUE means that the event has been fully taken into account and therefore will not propagated to other \textit{ObjectListener} objects (of parent classes).

Cf. Section 3.1.6.4, “Listeners Activation,” on page 3-37 for a detailed description of how cache, home and selection listeners are called.
3.1.6.3.7 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 ObjectFilter.

<table>
<thead>
<tr>
<th>Selection</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td>ObjectFilter</td>
</tr>
<tr>
<td>auto_refresh</td>
<td>boolean</td>
</tr>
<tr>
<td>members</td>
<td>ObjectRoot []</td>
</tr>
<tr>
<td>listener</td>
<td>SelectionListener</td>
</tr>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>attach_listener</td>
<td>void</td>
</tr>
<tr>
<td>detach_listener</td>
<td>void</td>
</tr>
<tr>
<td>refresh</td>
<td>void</td>
</tr>
</tbody>
</table>

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 ObjectFilter (filter); it is given at Selection creation time (cf. 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);
- list of the objects that are part of the selection (members);
- attached listener.

It offers the methods to:

- attach/detach a SelectionListener (attach_listener, detach_listener), that will be triggered when the composition of the selection changes, as well as if the members are modified;
- request that the Selection updates its members (refresh).
3.1.6.3.8 ObjectFilter

An ObjectFilter is an object (attached to a Selection) that gives the criterion to be applied to make the Selection.

<table>
<thead>
<tr>
<th><strong>ObjectFilter</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>check_object</td>
</tr>
<tr>
<td>an_object</td>
</tr>
<tr>
<td>filter</td>
</tr>
<tr>
<td>objects</td>
</tr>
</tbody>
</table>

It offers methods to:

- check if an object passes the filter – return value is TRUE – or not – return value is FALSE (check_object);
- filter a list of objects; the return value is the list of objects that return TRUE when checked with check_object (filter).

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

FooObjectFilter 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.

3.1.6.3.9 ObjectQuery

ObjectQuery is a specialization of ObjectFilter that perform the object check based on a query expression.

<table>
<thead>
<tr>
<th><strong>ObjectQuery</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>expression</td>
</tr>
<tr>
<td>parameters</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>set_query</td>
</tr>
<tr>
<td>expression</td>
</tr>
<tr>
<td>arguments</td>
</tr>
<tr>
<td>set_parameters</td>
</tr>
<tr>
<td>arguments</td>
</tr>
</tbody>
</table>

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). Cf. Appendix A 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.

### 3.1.6.3.10 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) then has to be implemented by the application, in order to be made aware of the incoming changes on objects belonging to a FooSelection.

<table>
<thead>
<tr>
<th>SelectionListener operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>on_object_in</td>
<td>void</td>
</tr>
<tr>
<td>on_object_out</td>
<td>void</td>
</tr>
<tr>
<td>on_object_modified</td>
<td>void</td>
</tr>
</tbody>
</table>

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 3.1.6.4, “Listeners Activation,” on page 3-37 includes a detailed description of how cache, home and selection listeners are called.
3.1.6.3.11 ObjectRoot

ObjectRoot is the abstract root for any DLRL class. It brings all the properties that are needed for DLRL management.

<table>
<thead>
<tr>
<th>ObjectRoot</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>oid</td>
</tr>
<tr>
<td>count</td>
</tr>
<tr>
<td>state</td>
</tr>
<tr>
<td>home</td>
</tr>
<tr>
<td>cache_access</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>destroy</td>
</tr>
<tr>
<td>is_modified</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>clean_modified</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>clone</td>
</tr>
<tr>
<td>access</td>
</tr>
<tr>
<td>scope</td>
</tr>
</tbody>
</table>

Its public attributes give:

- the identity of the object (oid);
- the number of times, it is referenced (count);
- its lifecycle state (state);
- its related home (home);
- the CacheAccess it belongs to (cache_access).

It offers methods to:

- create a copy\(^{14}\) of the object and attach it to a CacheAccess (clone_object); an object can be cloned to only to one CacheAccess allowing write operations; the operation takes as parameters the CacheAccess and the scope of the request (i.e., the object itself or the object and it components or the object and all the objects that are related);
- destroy itself;

\(^{14}\) The specification does not impose that the copy is performed at that moment; it just requires that it behaves ‘as if’. In particular, implementations may choose to actually copy the object only if needed (e.g., if incoming updates are coming for that object).
• see if the object has been modified by incoming updates (is_modified); is_modified takes as a 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);

• clean the modified status (clean_modified).

In addition, application classes (i.e., inheriting from ObjectRoot), will be generated with a set of methods dedicated to each shared attribute:

• 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);

• is_<attribute>_modified, to get if this attribute has been modified by means of incoming updates.

The object state is actually made of two parts:

• the write_state that is the state of the object with respect to write (and then publish) operations (on the following state diagram, set_xxx refers to a change of attribute):

\[ Figure 3-5 \quad \text{Write state of an object} \]

• the read_state that is the state of the object with respect to read (and incoming updates) operations.
For objects managed in both read and write modes (i.e., if the *Cache* usage is READ_WRITE):

- creating an object by the application (*create_object*), will init its *read state* as READ – the resulting *state* will therefore be CREATED/READ;
- creating an object, following an update will init its *write state* as WRITTEN – the resulting *state* will therefore be WRITTEN/NEW.

### 3.1.6.3.12 ObjectLink

Instances of ObjectLink classes are used to reference DLRL objects; they comprises the related *oid*, and a link to the corresponding *ObjectHome*.

<table>
<thead>
<tr>
<th><strong>ObjectLink</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>attributes</strong></td>
</tr>
<tr>
<td>oid</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td><strong>no operations</strong></td>
</tr>
</tbody>
</table>
3.1.6.3.13 Reference

Reference is a root class for typed references to objects.

<table>
<thead>
<tr>
<th>Reference : ObjectLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>object</td>
</tr>
<tr>
<td>ObjectRoot</td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td>ObjectRoot</td>
</tr>
<tr>
<td>is_modified</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>ReferenceScope</td>
</tr>
<tr>
<td>clean_modified</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>ReferenceScope</td>
</tr>
</tbody>
</table>

It offers methods to:

- modify the contents of the reference (put) end to retrieve the pointed object (get); these methods are generated for each concrete reference, to take into account the type of the value;
- see if the reference has been modified with incoming updates (is_modified); is_modified takes as parameter the scope of the request (i.e., the reference itself, or the reference and its referenced content);
- clean the modified status (clean_modified); it takes as parameter the scope of the request.

3.1.6.3.14 Collection

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

<table>
<thead>
<tr>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>length</td>
</tr>
<tr>
<td>integer</td>
</tr>
<tr>
<td>is_modified</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>ReferenceScope</td>
</tr>
<tr>
<td>clean_modified</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>scope</td>
</tr>
<tr>
<td>ReferenceScope</td>
</tr>
<tr>
<td>how_many_added</td>
</tr>
<tr>
<td>integer</td>
</tr>
<tr>
<td>how_many_removed</td>
</tr>
<tr>
<td>integer</td>
</tr>
<tr>
<td>removed_values</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>out: values</td>
</tr>
<tr>
<td>undefined</td>
</tr>
</tbody>
</table>

It provides the following methods:
• **length** to return the actual length of the collection;

• **is_modified** to return if the collection has been modified by incoming updates; **is_modified** takes as parameter the scope of the request (i.e., the collection itself, or the collection and its referenced content – note that this parameter makes a difference only if the collection contains objects);

• **clean_modified** to reset the modified status; it takes as parameter the scope of the request;

• **how_many_added** to return the number of added items;

• **how_many_removed** to return the number of removed items;

• **removed_values** to return the list of removed values, if manageable (a FALSE return value means that the collection is to be considered as fully modified).

3.1.6.3.15 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
```

<table>
<thead>
<tr>
<th>operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>void</td>
</tr>
<tr>
<td>index</td>
<td>integer</td>
</tr>
<tr>
<td>value</td>
<td>undefined</td>
</tr>
<tr>
<td>remove</td>
<td>void</td>
</tr>
<tr>
<td>get</td>
<td>undefined</td>
</tr>
<tr>
<td>index</td>
<td>integer</td>
</tr>
<tr>
<td>which_added</td>
<td>boolean</td>
</tr>
<tr>
<td>out: indexes</td>
<td>integer []</td>
</tr>
</tbody>
</table>

It provides the following methods:

• **put**, to put an item in the collection;

• **remove** to remove an item from the collection;

• **get** to retrieve an item in the collection (based on its index);

• **which_added** to return the indexes of the added objects, if manageable (a FALSE return value means that the collection is to be considered as fully modified).
3.1.6.3.16 **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 \texttt{FooStrMap}, assuming that \texttt{Foo} is the type of one item).

<table>
<thead>
<tr>
<th><strong>StrMap : Collection</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>get_all_keys</td>
</tr>
<tr>
<td>which_added</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>out: keys</td>
</tr>
</tbody>
</table>

It provides the following methods:

- \texttt{put}, to put an item in the collection;
- \texttt{remove} to remove an item from the collection;
- \texttt{get} to retrieve an item in the collection (based on its key);
- \texttt{get_all_keys} to retrieve all the keys of the items belonging to the map;
- \texttt{which_added} to return the keys of the added objects, if manageable (a specific return value means that the collection is to be considered as fully modified).

3.1.6.3.17 **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 \texttt{FooIntMap}, assuming that \texttt{Foo} is the type of one item).

<table>
<thead>
<tr>
<th><strong>IntMap : Collection</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
It provides the following methods:

- **put**, to put an item in the collection;
- **remove** to remove an item from the collection;
- **get** to retrieve an item in the collection (based on its key);
- **get_all_keys** to retrieve all the keys of the items belonging to the map;
- **which_added** to return the keys of the added objects, if manageable (a specific return value means that the collection is to be considered as fully modified).

### 3.1.6.3.18 Relation

**Relation** is the abstract root for all relations. It comprises an attribute that indicates whether the relation is a composition or not (**is_composition**)

<table>
<thead>
<tr>
<th>Relation</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>is_composition</td>
</tr>
<tr>
<td>no operations</td>
<td></td>
</tr>
</tbody>
</table>

### 3.1.6.3.19 RefRelation

**RefRelation** is the root for all generated classes that implement to-1 relations.

<table>
<thead>
<tr>
<th>RefRelation</th>
<th>: Relation, Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>no attributes</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>void</td>
</tr>
<tr>
<td>object</td>
<td>ObjectRoot</td>
</tr>
<tr>
<td>reset</td>
<td>void</td>
</tr>
</tbody>
</table>

It offers methods to modify the contents of the relation (set, reset); these methods are generated for each concrete relation, to take into account the type of the value as well as the inverse management if needed (in particular, they raise an exception (NotAllowed) if the inverse relation is a **MapRelation**);
3.1.6.3.20 ListRelation

*ListRelation* is the root class for all generated classes that implement to-n relations on a list basis.

<table>
<thead>
<tr>
<th>ListRelation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>values</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>object</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td>index</td>
</tr>
</tbody>
</table>

The methods to modify the contents of the *ListRelation* (*put, remove*) are generated to take into account the type of the value as well as the inverse management if needed. In particular, they raise an exception (NotAllowed) if the inverse relation is a *MapRelation*.

The method to retrieve one item (*get*) is generated to take into account the type of the value.

The *is_modified* definition that is applicable to a *ListRelation* is *Relation::is_modified*.

3.1.6.3.21 StrMapRelation

*StrMapRelation* is the root class for all generated classes that implement to-n relations on a map with string key basis.

<table>
<thead>
<tr>
<th>StrMapRelation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
</tr>
<tr>
<td>values</td>
</tr>
<tr>
<td>operations</td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td>key</td>
</tr>
<tr>
<td>object</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td>key</td>
</tr>
</tbody>
</table>

The methods to modify the contents of the *StrMapRelation* (*put, remove*) are generated to take into account the type of the value as well as the inverse management if needed.

The method to retrieve one item (*get*) is generated to take into account the type of the value.
The \texttt{is\_modified} definition that is applicable to a \texttt{StrMapRelation} is the one of \texttt{Relation::is\_modified}.

3.1.6.3.22 \texttt{IntMapRelation}

\texttt{IntMapRelation} is the root class for all generated classes that implement to-n relations on a map with integer key basis.

<table>
<thead>
<tr>
<th>\textbf{IntMapRelation}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{attributes}</td>
</tr>
<tr>
<td>\hspace{1cm} values</td>
</tr>
<tr>
<td>\textbf{operations}</td>
</tr>
<tr>
<td>\hspace{1cm} put</td>
</tr>
<tr>
<td>\hspace{1.5cm} key</td>
</tr>
<tr>
<td>\hspace{1cm} object</td>
</tr>
<tr>
<td>\hspace{1cm} remove</td>
</tr>
<tr>
<td>\hspace{1.5cm} get</td>
</tr>
<tr>
<td>\hspace{2cm} key</td>
</tr>
</tbody>
</table>

The methods to modify the contents of the \texttt{IntMapRelation} (\texttt{put}, \texttt{remove}) are generated to take into account the type of the value as well as the inverse management if needed.

The method to retrieve one item (\texttt{get}) is generated to take into account the type of the value.

The \texttt{is\_modified} definition that is applicable to a \texttt{IntMapRelation} is the one of \texttt{Relation::is\_modified}.

3.1.6.4 Listeners Activation

As described in Section 3.1.6.2, there are three kinds of listeners that the application developer may implement and attach to DLRL entities: \texttt{CacheListener}, \texttt{ObjectListener} and \texttt{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).

3.1.6.4.1 General Scenario

Incoming updates\textsuperscript{15} are usually a set of coherent individual updates that may be object creations, object deletions and object modifications.

\textsuperscript{15}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
This set of updates is managed as follows:

- First, the `CacheListener::start_updates` operation is triggered.
- Then all the updates are actually applied in the cache\(^\text{16}\). When an object is modified, several operations allow to get more precisely which parts of the object are concerned (cf. `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 (cf. the following sections).
- Finally the `CacheListener::end_updates` operation is triggered and the modification states of the updated object is cleaned (if not already done).

### 3.1.6.4.2 Object Creation

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

- First, an attempt is made to find the suitable `ObjectListener` and to trigger its `on_object_created` operation; the suitable `ObjectListener` is defined as the first `ObjectListener`, following the inheritance structure from the exact home (e.g., `FooHome`, if the new object is an instance of `Foo`) to the root `ObjectRootHome`, whose `on_object_created` returns true.
- Then, all the `Selection` objects that are concerned with that kind of objects (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.

### 3.1.6.4.3 Object Modification

When an individual update reports an object modification, 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 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.

\(^{16}\)If an object is deleted, its state is set as DELETED; it will be actually removed when there are no more references to it.
Then, an attempt is made to find the suitable ObjectListener and to trigger its on_object_modified operation; the suitable ObjectListener is defined as the first ObjectListener, following the inheritance structure from the exact home (e.g., FooHome, if the new object is an instance of Foo) to the root ObjectRootHome, whose on_object_modified returns true.

### 3.1.6.4.4 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, an attempt is made to find the suitable ObjectListener and to trigger its on_object_deleted operation; the suitable ObjectListener is defined as the first ObjectListener, following the inheritance structure from the exact home (e.g., FooHome, if the new object is an instance of Foo) to the root ObjectRootHome, whose on_object_deleted returns true.

### 3.1.6.5 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 CacheUsage per thread, DLRL does not enforce this rule by any means.

#### 3.1.6.5.1 Read Mode

The typical scenario for read mode is as follows:

1. Create the CacheAccess for read purpose (Cache::create_access);
2. Clone some objects in it (ObjectRoot::clone);
3. Refresh them (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;

#### 3.1.6.5.2 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);
3. Refresh them (\texttt{CacheAccess::refresh});

4. If needed create new ones for that \texttt{CacheAccess (ObjectHome:: create_object)};

5. Modify the attached (plain access to the objects);

6. Write the modifications into the underlying infrastructure (\texttt{CacheAccess::write});

7. Purge the cache (\texttt{CacheAccess::purge}); step 2 can be started again;

8. Eventually, delete the \texttt{CacheAccess (Cache::delete_cache)}.

\subsection*{3.1.6.6 Generated Classes}

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

- \texttt{FooHome} : \texttt{ObjectHome}
- \texttt{FooListener} : \texttt{ObjectListener}
- \texttt{FooSelection} : \texttt{Selection}
- \texttt{FooSelectionListener} : \texttt{SelectionListener}
- \texttt{FooFilter} : \texttt{ObjectFilter}
- \texttt{FooQuery} : \texttt{FooFilter, ObjectQuery}

- And for relations to \texttt{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):
  - "\texttt{FooRelation}" : \texttt{RefRelation}
  - "\texttt{FooListRelation}" : \texttt{ListRelation}
  - "\texttt{FooStrMapRelation}" : \texttt{StrMapRelation}
  - "\texttt{FooIntMapRelation}" : \texttt{IntMapRelation}

\subsection*{3.2 CORBA 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 3.2.1, “CORBA Run-time Entities,” on page 3-41.

This section also specifies the generation process (specializing the abstract one presented on Figure 25: DLRL Generation Process); in particular, the following are described:

- How the application introduces its application classes ("Model Description" on Figure 25);
- How the application adds indication to properly generate the DLRL entities as well as the resulting enhanced application constructs ("Model Tags" on Figure 25).

This process is described in Section 3.2.2, “Generation Process,” on page 3-55.
3.2.1 CORBA Run-time Entities

3.2.1.1 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 modelled as IDL local interfaces; entities belonging to the second one are modelled as IDL valuetypes (with the exception of ObjectLink, which is a fixed-size structure, so that it can be embedded).

The choice for valuetypes has been driven by two main reasons:
- It is the IDL construct that fits the 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.

3.2.1.2 IDL Description

This IDL is split in two sections:
- IDL for the generic DLRL entities;
- Implied IDL.

3.2.1.2.1 Generic DLRL Entities

#include "dcps.idl"

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

module Dlrl {

// 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
};

// State of an object
// ------------------
typedef unsigned short ObjectState;

// Read part of the state
const ObjectState R_NEW = 0x0001 << 0;
const ObjectState R_MODIFIED = 0x0001 << 1;
const ObjectState R_READ = 0x0001 << 2;
const ObjectState R_DELETED = 0x0001 << 3;

// Write part of the state
const ObjectState W_CREATED = 0x0001 << 8;
const ObjectState W_CHANGED = 0x0001 << 9;
const ObjectState W_WRITTEN = 0x0001 << 10;
const ObjectState W_DESTROYED = 0x0001 << 11;
const ObjectState W_DELETED = 0x0001 << 12;
// OID
// ----

typedef unsigned long DLRLOid;

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

typedef sequence<string> stringSeq;
typedef sequence<long> longSeq;

// Exceptions
// ============

description DCPSError {};
description BadHomeDefinition {};
description BadParameter {};
description NotFound {};
description ReadOnlyMode {};
description WriteOnlyMode {};

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

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

valuetime ObjectRoot;
typedef sequence<ObjectRoot> ObjectRootSeq;
local interface ObjectHome;
typedef sequence<ObjectHome> ObjectHomeSeq;
local interface Selection;
typedef sequence<Selection> SelectionSeq;
local interface CacheAccess;
typedef sequence<CacheAccess> CacheAccessSeq;
local interface Cache;

************
* ObjectLink
************

struct ObjectLink {
    DLRLOid oid;
}
unsigned long home_index;
};
typedef sequence<ObjectLink> ObjectLinkSeq;

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

local interface ObjectListener {
/***
 * will be generated with the proper Foo type
 * in the derived FooObjectListener
 *
 boolean on_object_created (in ObjectRoot the_object);
 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 begin_updates (
    in long update_round);
  void end_updates (
    in long update_round);
};

/******************************************
* ObjectRoot : Root fot the shared objects
******************************************/

local interface ObjectRootOperations {

// Attributes
// ----------
readonly attribute DLRLOid oid;
readonly attribute ObjectState state;
readonly attribute unsigned short count;
readonly attribute ObjectHome owner;
readonly attribute CacheAccess cache;

// Operations
// ----------
void destroy ()
  raises (DCPSError,
    ReadOnlyMode);
boolean is_modified (in ObjectScope scope);
void clean_modified (in ObjectScope scope);
ObjectLink clone (in CacheAccess access,
  in ObjectScope scope)
  raises (ReadOnlyMode);
};

valuetype ObjectRoot supports ObjectRootOperations {
  private DLRLOid m_oid;
  private string m_class_name;
};

/**********************************************
* ObjectFilter: Root of all the objects filters
**********************************************/

local interface ObjectFilter {

  /***
   * Following method will be generated properly typed

   ***/

  boolean is_modified (in ObjectScope scope);
};
* in the generated derived classes
*
boolean check_object (
in ObjectRoot an_object);
ObjectRootSeq filter (
in ObjectRootSeq objects);
*
***/
};

/*******************************************************************************/
* ObjectQuery : Specialization of the above to make a Query
*******************************************************************************/

local interface ObjectQuery {

// Attributes
// -------
readonly attribute string expression;
readonly attribute stringSeq parameters;

//--- Methods
boolean set_query (in string expression,
in stringSeq parameters);
boolean set_parameters (in stringSeq parameters);
};

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

local interface Selection {

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

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

// Operations
/**
 * Following method will be generated properly typed
 * in the generated derived classes
 *
 * void attach_listener (
 *     in SelectionListener listener);
 * 
 * ***/
 * void detach_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 filter;
    readonly attribute ObjectScope notification_scope;
    readonly attribute ObjectHome parent;
    readonly attribute ObjectHomeSeq children;

    /**
     * Following attributes will be generated properly typed
     * in the generated derived classes
     *
     * readonly attribute ObjectRootSeq extent;
     * readonly attribute SelectionSeq selections;
     * readonly attribute ObjectListener listener;
     * 
     * ***/
     */

    // Operations
    // ---------
    string get_topic_name (  
        in string attribute_name)
     raises (  
            BadParameter);
    void set_filter (  
        in string filter)
     raises (  
            BadParameter);

    // --- Listener management
/** *
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
 * void attach_lister (  
 *     in ObjectListener listener);
 * 
 * ***/
 * void detach_listener ();
 *
 */

// --- Selection management

/***  
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
 * Selection create_selection (  
 *     in ObjectFilter filter,  
 *     in boolean auto_refresh)  
 * raises (  
 *     BadParameter);  
 * void delete_selection (  
 *     in Selection a_selection)  
 * raises (  
 *     BadParameter);  
 * 
 * ***/
 *
// --- Object management

/***  
 * Following methods will be generated properly typed
 * in the generated derived classes
 *
 * ObjectRoot create_object(  
 *     in CacheAccess access)  
 * raises (  
 *     ReadOnlyMode);
 * ObjectRoot find_object (  
 *     in DLRLOid oid,  
 *     in CacheAccess access)  
 * raises (  
 *     NotFound);  
 * 
 * ***/
 * 
 */

/****************************/  
* Reference and Collection operations
***************************************************************************/
local interface ReferenceOperations {
    void reset();
    boolean is_modified (
        in ReferenceScope scope);
    void clean_modified ( 
        in ReferenceScope scope);
};

local interface CollectionOperations {
    long length();
    boolean is_modified ( 
        in ReferenceScope scope);
    long how_many_added ();
    long how_many_removed ();
    void clean_modified ( 
        in ReferenceScope scope);
};

local interface ListOperations : CollectionOperations {
    longSeq which_added ();
    void remove ();
};

local interface StrMapOperations : CollectionOperations {
    stringSeq which_added ();
    stringSeq get_all_keys ();
};

local interface IntMapOperations : CollectionOperations {
    longSeq which_added ();
    longSeq get_all_keys ();
};

/***************************
* Value Bases for Relations
***************************/

valuetype RefRelation supports ReferenceOperations {
    private ObjectLink ref;
};

valuetype ListRelation supports ListOperations {
    private ObjectLinkSeq refs;
};

valuetype StrMapRelation supports StrMapOperations {
    struct Item {
        string key;
        ObjectLink ref;
    }
typedef sequence <Item> ItemSeq;

private ItemSeq refs;
};

valuetype IntMapRelation supports IntMapOperations {
    struct Item {
        long key;
        ObjectLink ref;
    };
    typedef sequence <Item> ItemSeq;

    private ItemSeq refs;
};

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

local interface CacheAccess {

    // Attributes
    // =========
    readonly attribute CacheUsage cache_usage;
    readonly attribute Cache owner;
    readonly attribute ObjectLinkSeq refs;

    // Operations
    // =========
    void refresh ()
        raises (DCPSError);
    void write ()
        raises (ReadOnlyMode, DCPSError);
    void purge ();
    ObjectRoot deref (in ObjectLink ref);
};

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

local interface Cache : CacheAccess {
// Attributes
// -----------
readonly attribute DCPSState pubsub_state;
readonly attribute DCPS::Publisher publisher;
readonly attribute DCPS::Subscriber subscriber;
readonly attribute boolean updates_enabled;
readonly attribute ObjectHomeSeq homes;
readonly attribute CacheAccessSeq sub_accesses;
readonly attribute CacheListener listener;

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

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

//-- Home management
void register_home (in ObjectHome a_home) raises (BadHomeDefinition);
ObjectHome find_home (in string class_name) raises (NotFound);

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

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

//-- CacheAccess Management
CacheAccess create_access (in CacheUsage purpose) raises (ReadOnlyMode);
void delete_cache (in CacheAccess access) raises (BadParameter);
// --- Object management
void load ()
  raises (DCPSError);
};

/*****************************************
* CacheFactory : Factory to create Cache objects
*****************************************/
local interface CacheFactory {
  Cache create_cache (in CacheUsage cache,
                      in DCPS::DomainParticipant domain)
    raises (DCPSError);
};

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

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

/*************************/
* DERIVED CLASSES FOR Foo
*************************/
typedef sequence<Foo> FooSeq;

local interface FooListener : DLRL::ObjectListener {
  boolean on_object_created (in Foo the_object);
  boolean on_object_modified (in Foo the_object);
  boolean on_object_deleted (in Foo the_object);
};

local interface FooSelectionListener : DLRL::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);
};
in Foo the_object);
};

local interface FooFilter : DLRL::ObjectFilter {
  boolean check_object (in Foo an_object);
  FooSeq filter (in FooSeq objects);
};

local interface FooQuery : DLRL::ObjectQuery, FooFilter {
};

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

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

local interface FooHome: DLRL::ObjectHome {
  readonly attribute FooSeq extent;
  readonly attribute FooSelectionSeq selections;
  readonly attribute FooListener listener;
  void attach_listener (in FooListener listener);
  FooSelection create_selection (in FooFilter filter,
                             in boolean auto_refresh)
                             raises (DLRL::BadParameter);
  void delete_selection (in FooSelection a_selection)
                         raises (DLRL::BadParameter);
  Foo create_object(in DLRL::CacheAccess access)
                         raises (DLRL::ReadOnlyMode);
  Foo find_object (in DLRL::DRLROid oid,
                       in DLRL::CacheAccess access)
                         raises (DLRL::NotFound);
};
/********************************************
* Derived class for relations to Foo
********************************************/

valuetype FooRef : DLRL::RefRelation { // Ref<Foo>
   void set(
      in Foo object);
};

valuetype FooList : DLRL::ListRelation { // List<Foo>
   void put (  
      in long index,  
      in Foo a_foo);
   Foo get (  
      in long index)  
      raises (  
      DLRL::NotFound);
};

valuetype FooStrMap : DLRL::StrMapRelation { // StrMap<Foo>
   void put ( 
      in string key,  
      in Foo a_foo);
   Foo get (  
      in string Key)  
      raises (  
      DLRL::NotFound);
   void remove (  
      in string Key);
};

valuetype FooIntMap : DLRL::IntMapRelation { // IntMap<Foo>
   void put (  
      in long key,  
      in Foo a_foo);
   Foo get (  
      in long Key)  
      raises (  
      DLRL::NotFound);
   void remove (  
      in long Key);
};
3.2.2 Generation Process

3.2.2.1 Principles

The generic generation process explained in Section 3.1.4.5, “How is this Mapping Indicated?,” on page 3-12, is instantiated as follows:

![Diagram of DLRL Generation Process (CORBA)](image)

**Figure 3-7** DLRL Generation Process (CORBA)

3.2.2.2 Model Description

As stated in Section 3.2.1, “CORBA Run-time Entities,” on page 3-41, application classes are modelled 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.

3.2.2.3 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.

3.2.2.3.1 Model Tags DTD

The following is the DTD for expressing the Model Tags in XML:
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT Dirl (enumDef | templateDef | associationDef | compoRelationDef | classMapping)*>
<!ATTLIST Dirl name CDATA #IMPLIED>
<!ELEMENT enumDef (value)*>
<!ATTLIST enumDef name CDATA #REQUIRED>
<!ELEMENT value (#PCDATA)>
<!ELEMENT templateDef EMPTY>
<!ATTLIST templateDef name CDATA #REQUIRED>
<!ELEMENT associationDef (relation, relation)>
<!ATTLIST relation class CDATA #REQUIRED>
<!ATTLIST compoRelationDef class CDATA #REQUIRED>
<!ELEMENT compoRelationDef EMPTY>
<!ATTLIST compoRelationDef class CDATA #REQUIRED>
<!ELEMENT classMapping (mainTopic, extensionTopic?, (monoAttribute | multiAttribute | monoRelation | multiRelation | local))*>
<!ATTLIST classMapping name CDATA #REQUIRED>
<!ELEMENT mainTopic EMPTY>
<!ATTLIST mainTopic name CDATA #REQUIRED>
<!ELEMENT extensionTopic EMPTY>
<!ATTLIST extensionTopic name CDATA #REQUIRED>
<!ELEMENT monoAttribute (placeTopic?, valueField+)>
<!ATTLIST monoAttribute name CDATA #REQUIRED>
<!ELEMENT multiAttribute (multiPlaceTopic, valueField+)>
<!ATTLIST multiAttribute name CDATA #REQUIRED>
<!ELEMENT monoRelation (placeTopic?, valueKey)>
<!ATTLIST monoRelation name CDATA #REQUIRED>
<!ELEMENT multiRelation (multiPlaceTopic, valueKey)>
<!ATTLIST multiRelation name CDATA #REQUIRED>
<!ELEMENT local EMPTY>
<!ATTLIST local name CDATA #REQUIRED>
<!ELEMENT valueKey EMPTY>
<!ATTLIST valueKey classField CDATA #IMPLIED>
<!ELEMENT placeTopic EMPTY>
<!ATTLIST placeTopic name CDATA #REQUIRED>
<!ELEMENT multiPlaceTopic EMPTY>
<!ATTLIST multiPlaceTopic name CDATA #REQUIRED>
<!ELEMENT valueField (#PCDATA)>
3.2.2.3.2 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.

3.2.2.3.2.1 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).

3.2.2.3.2.2 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>
```

3.2.2.3.2.3 TemplateDef

This tag contains three attributes

- `name`, that gives the scoped name of the type
- `pattern`, that gives the collection pattern (are supported List, StrMap and IntMap);
- `itemType`, that gives the type of each element in the collection.
Example:
<templateDef name="BarStrMap" basis="StrMap" itemType="Bar"/>

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

3.2.2.3.2.4 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**, that contains the scoped name of the class;
- **attribute**, that 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>

3.2.2.3.2.5 compoRelationDef

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

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

Example:
<compoRelationDef class="Radar" attribute="tracks"/>

3.2.2.3.2.6 ClassMapping

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

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

Example:
<classMapping name="Track">
  ...
</classMapping>

3.2.2.3.2.7 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 the following attributes:

- **name** which is the name of the Topic;
• **classField** which is the name of the field that will contains the class name (if needed – cf. Section 3.1.4.2.2);

• **oidField**, which is the name of the field that will contain the oid.

Example:
```xml
<mainTopic name="TRACK-TOPIC" classField="CLASS" oidField="OID"/>
```

3.2.2.3.2.8 *ExtensionTable*

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

Example:
```xml
<extensionTopic name="TRACK3D-TOPIC" classField="CLASS" oidField="OID"/>
```

3.2.2.3.2.9 *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 to give the name of the field(s) that will contain the value of that attribute.

Example:
```xml
<monoAttribute name="y">
  <placeTopic name="Y_TOPIC" classField="CLASS" oidField="OID"/>
  <valueField>Y</valueField>
</monoAttribute>
```

3.2.2.3.2.10 *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 to give the name of the field(s) that will contain the value of that attribute.

Example:
```xml
<multiAttribute name="comments">
  <multiPlaceTopic name="COMMENTS-TOPIC" classField="CLASS" oidField="OID" indexField="INDEX"/>
  <valueField>COMMENT</valueField>
</multiAttribute>
```
3.2.2.3.2.11 MonoRelation

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

- a mandatory attribute to give the **name** of the attribute;
- an optional sub-tag to give the **Topic** where it is placed (**placeTopic** – cf. Section 3.2.2.3.2.9);
- one **valueKey** 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:

```xml
<monoRelation name="a_radar">
  <valueKey oidField="RADAR_OID"/>
</monoRelation>
```

3.2.2.3.2.12 MultiRelation

This tag gives the mapping for a multi-valued relation. Its 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** – cf. Section 3.2.2.3.2.10);
- one **valueKey** sub-tag (cf. Section 3.2.2.3.2.11).

Example:

```xml
<multiRelation name="tracks">
  <multiPlaceTopic name="RADARTRACKS-TOPIC" oidField="RADAR-OID" indexField="INDEX"/>
  <valueKey classField="TRACK-CLASS" oidField="TRACK-OID"/>
</multiRelation>
```

3.2.2.3.2.13 Local

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

Example:

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

3.2.3 Example

This section contains a very simple example, to illustrate DLRL.
### 3.2.3.1 UML Model

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

![UML Class Diagram of the Example](image)

Figure 3-8  UML Class Diagram of the Example

### 3.2.3.2 IDL Model Description

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

```idl
#include "dlrl.idl"

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

valuetype Track : DLRL::ObjectRoot {
    public double       x;
    public double       y;
    public stringStrMap comments;
    public long         w;
}

valuetype Track3D : DLRL::ObjectRoot {
    public double       x;
    public double       y;
    public double       z;
}
```

### 3.2.3.3 XML Model Tags

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

```xml
<?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"/>
    <templateDef name="RadarRef" pattern="Ref" itemType="Radar"/>
    <classMapping name="Track">
        Track
        x : real
        y : real
        comments [*] : string
        w : integer
    </classMapping>
    <classMapping name="Track3D">
        Track3D
        x : real
        y : real
        z : real
    </classMapping>
</Dlrl>
```
It should be noted that XML is not suitable for manual edition, therefore the file seems much more complicated than it actually is; it seems much simpler when viewed through an XML editor, as on the following picture illustrates.

Figure 3-9  XML Editor Illustration

Also note that only the three `templateDef`, the `associationDef` and the `local`\textsuperscript{17} tags are mandatory in all cases: the `ClassMapping` tags are only required if a deviation is wanted from the default mapping described in Section 3.1.4.3. In this case, the XML file would be as follows:
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Drl SYSTEM "dlrl.dtd">
<Drl name="Example">
  <templateDef name="stringStrMap" pattern="StrMap" itemType="string"/>
  <templateDef name="RadarRef" pattern="Ref" itemType="Radar"/>
  <templateDef name="TrackList" pattern="List" itemType="Track"/>
  <classMapping name="Track">

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

### 3.2.3.4 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:

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

<table>
<thead>
<tr>
<th>Y-TOPIC</th>
<th>Topic to store Track::y, outside Track’s main topic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>Field to store the class part of the object reference</td>
</tr>
<tr>
<td>OID</td>
<td>Field to store the oid part of the object reference</td>
</tr>
<tr>
<td>Y</td>
<td>Field to store the value of the attribute y</td>
</tr>
</tbody>
</table>

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

18. 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.
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.

### 3.2.3.5 Code Example

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

```cpp
dcps::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);
t3->z(3000.0);
t2->a-radar->put(r1);// modifies r1->tracks accordingly
c->write();// all modifications are published
};
Compliance Points

This specification includes the following compliance profiles.

- **Minimum profile**: This profile contains just the mandatory features of the DCPS layer. None of the optional features are included.

- **Content-subscription profile**: This profile adds the optional classes: `ContentFilteredTopic`, `QueryCondition`, `MultiTopic`. This profile enables subscriptions by content. See page 2 of the specification.

- **Persistence profile**: This profile adds the optional setting ‘PERSISTENT’ of the DURABILITY QoS policy. This profile enables saving data into permanent storage so that it can survive system outages. See page 2 of the specification.

- **Ownership profile**: This profile adds two things: First the optional setting ‘EXCLUSIVE’ of the OWNERSHIP kind. Second support for the optional OWNERSHIP_STRENGTH policy. Third the ability to set a `depth > 1` for the HISTORY QoS policy.

- **Object model profile**: This profile includes the DLRL and also includes support for the PRESENTATION `access_scope` setting of ‘GROUP’ (page 2 of the specification).
Syntax for DCPS Queries and Filters

A subset of SQL syntax is used in several parts of the specification:

- the *filter_expression* in the *ContentFilteredTopic* (cf. “ContentFilteredTopicClass” on page 2-29).
- the *query_expression* in the *QueryReadCondition* (cf. “QueryCondition Class” on page 2-60).

Those expressions may use a subset of SQL, extended with the possibility to use program variables in the SQL expression. The allowed SQL expressions are defined with the BNF-grammar below.

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*.

SQL grammar in BNF

```plaintext
Expression ::= FilterExpression
  | TopicExpression
  | QueryExpression

FilterExpression ::= Condition

TopicExpression ::= SelectFrom {Where } ';'

QueryExpression ::= {Condition}{'ORDER BY' (FIELDNAME // ',' )}

SelectFrom ::= 'SELECT' Aggregation 'FROM' Selection
```


Aggregation ::= `*`  
  | (SubjectFieldSpec // `,`)  
  
SubjectFieldSpec ::= FIELDNAME  
  | FIELDNAME `AS` FIELDNAME  
  | FIELDNAME FIELDNAME  
  
Selection ::= TOPICNAME  
  | TOPICTNAME NaturalJoin JoinItem  
  
JoinItem ::= TOPICNAME  
  | TOPICNAME NaturalJoin JoinItem  
  | `(' TOPICNAME NaturalJoin JoinItem `)’  
  
NaturalJoin ::= `INNER NATURAL JOIN`  
  | `NATURAL JOIN`  
  | `NATURAL INNER JOIN`  
  
Where ::= `WHERE` Condition  
  
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 ::= `=` | `>` | `>=` | `<` | `<=` | `<>` | `like`  
  
Range ::= Parameter `AND` Parameter  
  
Parameter ::= INTEGERVALUE  
  | FLOATVALUE  
  | STRING  
  | PARAMETER  
  
Note – INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. They are all supported because they all are part of the SQL standard.

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 FIELDNAME can refer to fields at any 
  depth in the data structure.

• TOPICNAME - A topic name is an identifier for a topic, and is defined as any 
  series of characters `a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`, `-` but may not 
  start with a digit.

• 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.

• 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`, 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.

• 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.

Examples

Assuming Topic “Location” has as an associated type a structure with fields 
“flight_name, x, y, z”, and Topic “FlightPlan” has as fields “flight_id, source, 
destination”. The following are examples of using these expressions.

Example of a **topic_expression**:

- “SELECT flight_name, x, y, z AS height FROM ‘Location’ NATURAL JOIN 
  ‘FlightPlan’ WHERE height < 1000 AND x <23”

Example of a **query_expression** or a **filter_expression**:

- “height < 1000 AND x <23”
Syntax for DLRL Queries and Filters

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 **ObjectHome** (cf. “ObjectHome” on page 4-23)
- the **query** in the **ObjectQuery** (cf. “ObjectQuery” 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 .
```
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.

- **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.

- **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 en, 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.

- **PARAMETER** - A parameter is of the form %n, where n represents a natural number (zero included) smaller than 100. It refers to the n + 1th argument in the given context.