Tutorial on the OMG Data Distribution Service

Victor Giddings
Objective Interface Systems, Inc.
victor.giddings@ois.com

Introduction
OMG Data Distribution Service

- Adopted by OMG in June 2003
- Joint submission of
  - Real-Time Innovations
  - THALES
  - Objective Interface Systems
  - MITRE (Supporter)
- Finalized in April 2004
- Current version – 1.1 – March 2005
- Provides data distribution services
  - Typed
  - Multi-point
  - Scalable
  - Quality of Service (QoS)-controlled
- Two layers
  - Data Centric Publish and Subscribe – DCPS
  - Data Local Reconstruction Layer - DLRL

Specification Contents

- **Overview**
  - 1.1 Introduction
  - 1.2 Purpose
- **Data-Centric Publish- Subscribe (DCPS)**
  - 2.1 Platform Independent Model (PIM)
    - 2.1.1 Overview and Design Rationale
    - 2.1.2 PIM Description
    - 2.1.3 Supported QoS
    - 2.1.4 Listeners, Conditions and Wait-sets
    - 2.1.5 Built-in Topics
    - 2.1.6 Interaction Model
  - 2.2 OMG IDL Platform Specific Model (PSM)
    - 2.2.1 Introduction
    - 2.2.2 PIM to PSM Mapping Rules
    - 2.2.3 DCPS PSM : IDL

Specification Contents

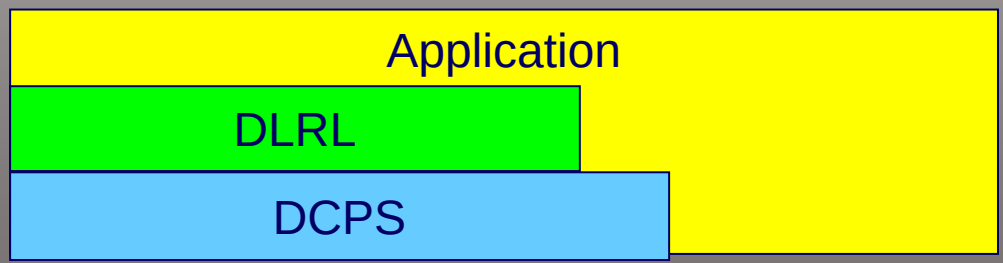
- **Data Local Reconstruction Layer (DLRL)**
  - 3.1 Platform Independent Model (PIM)
    - 3.1.1 Overview and Design Rationale
    - 3.1.2 DLRL Description
    - 3.1.3 What Can Be Modeled with DLRL
    - 3.1.4 Structural Mapping
    - 3.1.5 Operational Mapping
    - 3.1.6 Functional Mapping
  - 3.2 OMG IDL Platform Specific Model (PSM)
    - 3.2.1 Run-time Entities
    - 3.2.2 Generation Process
    - 3.2.3 Example
- **A - Compliance Points**
- **B - Syntax for DCPS Queries and Filters**
- **C - Syntax for DLRL Queries and Filters**

What are PIM and PSM?

- Model Driven Architecture
  - Transformation of one model to another
  - Model is a suitable abstraction
- In Model Driven Architecture terminology
  - PIM – Platform Independent Model
    - More abstract model – semantics without specific syntax
    - In DDS specification – PIM is specified in Universal Modelling Language (UML)
  - PSM – Platform Specific Model
    - Concrete representation – specific syntax
    - In DDS specification
      - Only one PSM specified: OMG IDL
      - Mapping rules specified
- In the rest of this tutorial – ignore PIM/PSM separation; cover syntax and semantics together

Architecture of DDS Specification

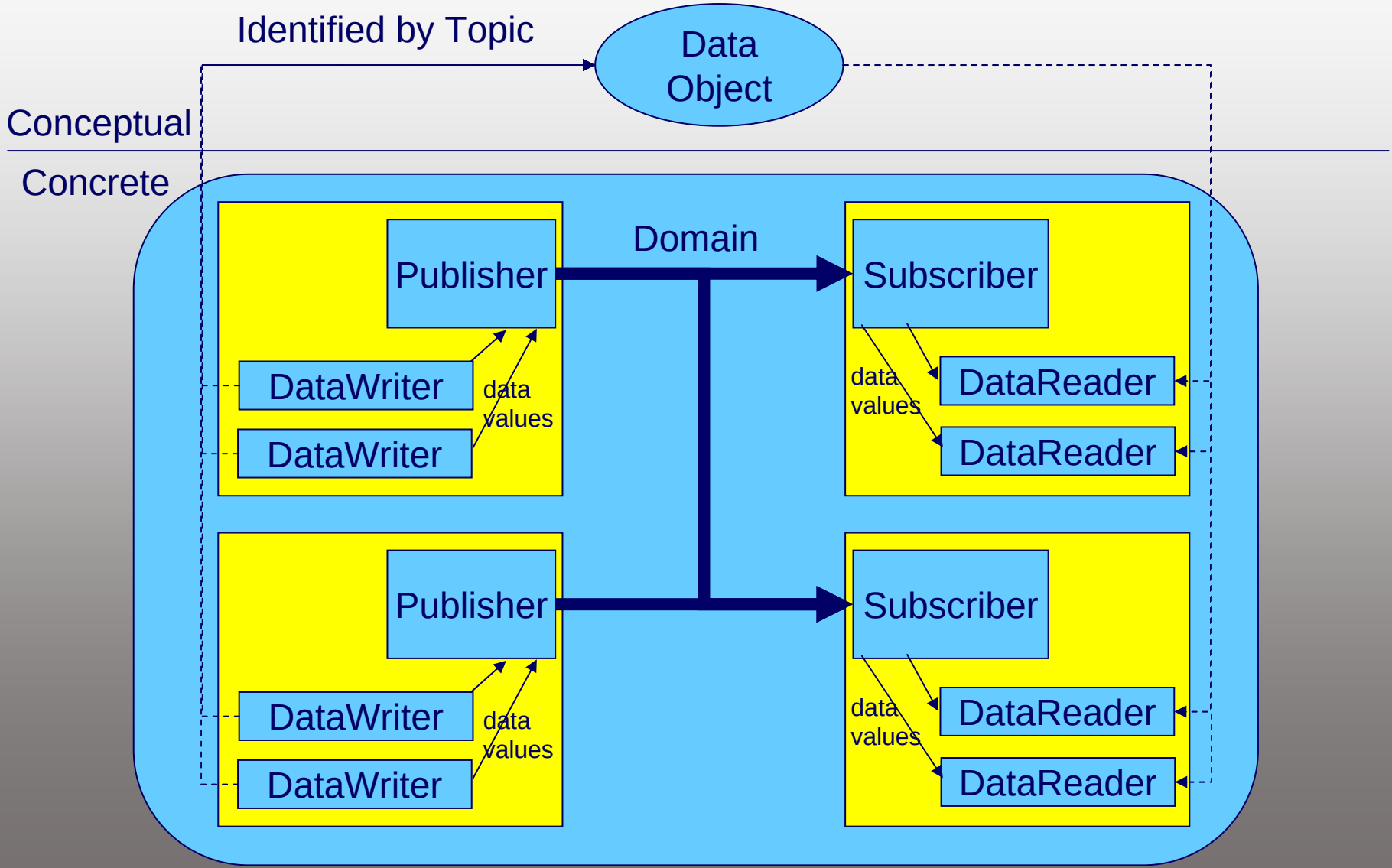
- Data-Centric Publish-Subscribe (DCPS)
  - Lower level
  - Targeted towards efficient delivery of the proper information to the proper recipients
- Data Local Reconstruction Layer (DLRL)
  - Optional higher level
  - Automatically reconstructs data locally from distributed updates
  - Allows the application to access the data ‘as if’ it were local
  - Propagates information to all interested subscribers but also updates a local copy of the information.



Data Centric Publish-Subscribe (DCPS)

Data Distribution Service - Part I

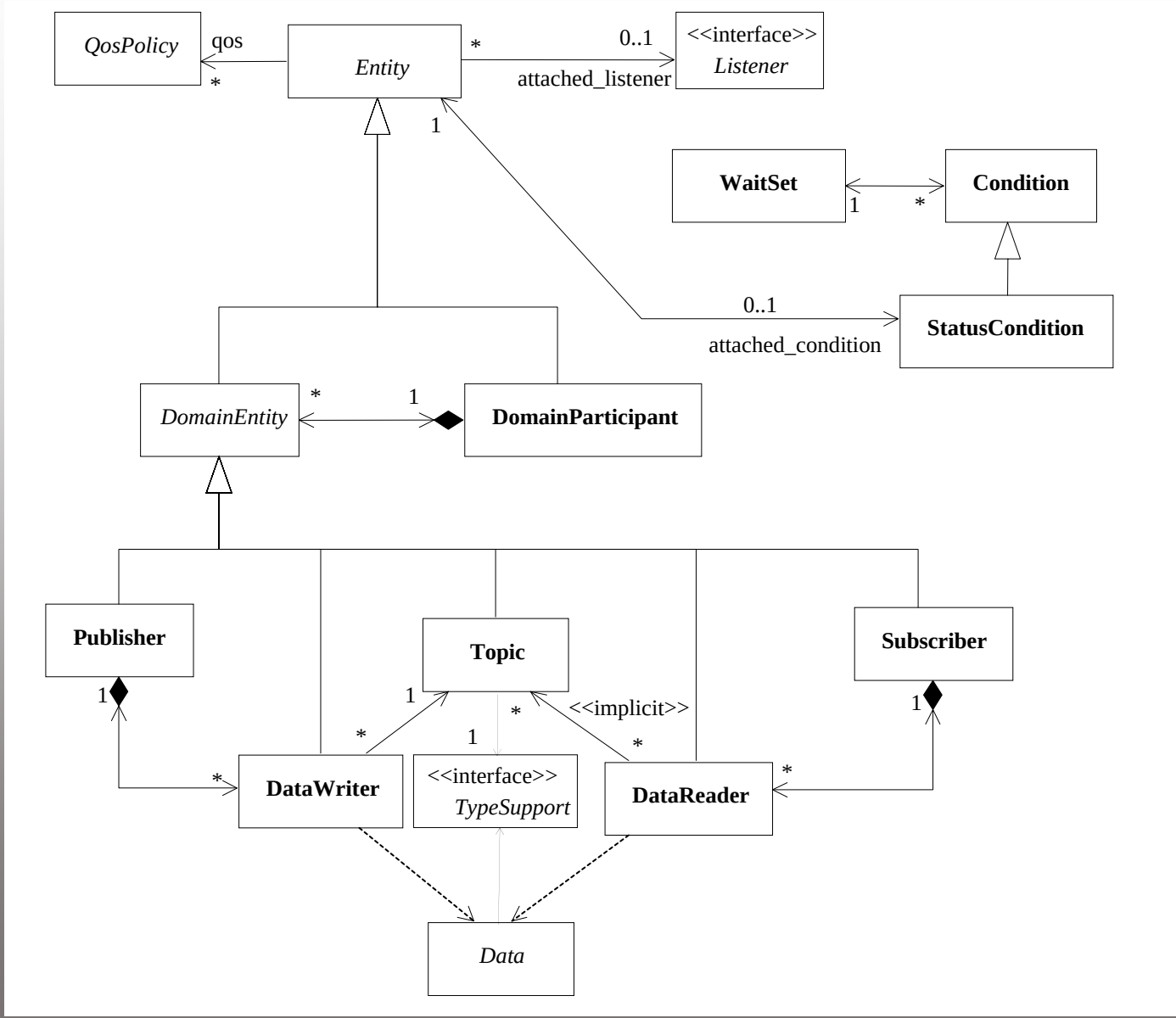
DCPS Data Flow Architecture



DCPS Data Flow Entities

- Publisher
  - Responsible for data dissemination
  - Publishes data of different types
- DataWriter
  - Communicates to a publisher the existence and value of data-objects
  - Typed access to Publisher
- Subscriber
  - Responsible for receiving published data
  - Receives data of different types
- DataReader
  - Typed access to Subscriber
  - Provides data-values to application
- Topic – identifies typed data flow

DCPS UML Model



Other DCPS Entities

- Entity – abstract base class
- QoSPolicy – abstract base class to hold Quality of Service settings
- Listener
  - Abstract base class
  - Allows reaction to changes
  - May be attached to DCPS Entity
- Condition
  - Represents something that can be waited for
  - Abstract base class
  - Specializations: StatusCondition, GuardCondition, ReadCondition
- WaitSet – set of conditions to be waited for

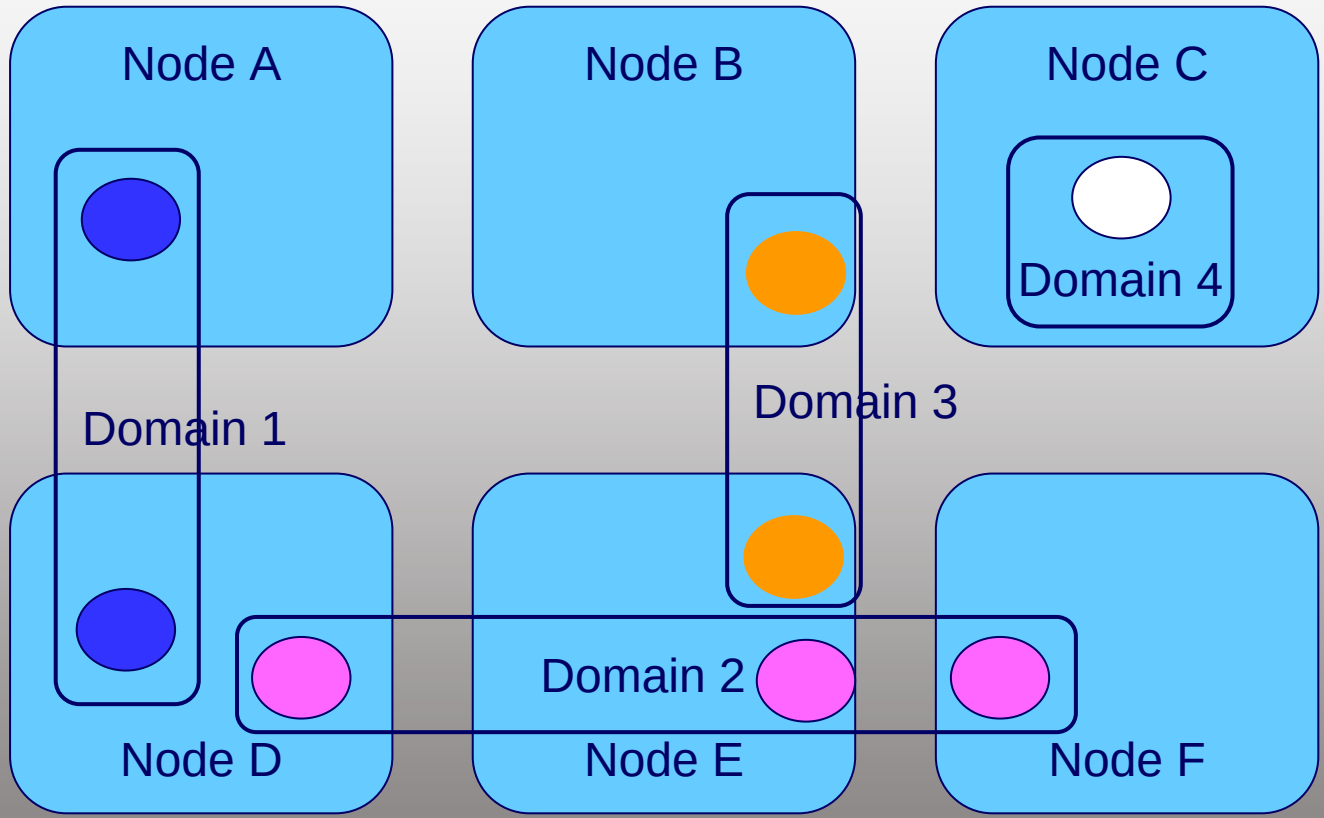
Other DCPS Entities

- Status – predefined state of communication entity
- StatusCondition
  - A state that can be waited for
  - Attached to an entity
- DomainParticipant
  - Represents the local membership of the application in a *domain.*
- DomainEntity
  - Abstract intermediate class
  - Ensures a DomainParticipant cannot contain another
- TypeSupport – registration of data type names.

DCPS Domains

- A *domain* links all the applications able to communicate with each other
- Only the publishers and the subscribers attached to the same *domain* may interact
- Domains and *nodes* (computers) are orthogonal
  - One node may participate in multiple domains
  - Domains may span a single or multiple nodes
- One DomainParticipant on each node for each domain that it participates in
- Topic names are unique within domain

DCPS Domains



DCPS Topics

- Topics provide the identification of the data that
  - Publishers provide
  - Subscribers receive
- Identified by a string name
- Must be unique within domain
- Associated with a single *type* of data
  - Expressed in OMG Interface Definition Language (IDL)
  - Type name registered with DCPS with `register_type`
  - Associated by type name with `create_topic`
- Represented by DCPS class `Topic`

DCPS Data and DataTypes

- Data types represent information that is sent atomically
  - DLRL may be used to break down objects into elements
- By default, each data modification is propagated individually, independently, and uncorrelated with other modifications
- Application may request that several modifications be sent as a whole and interpreted as such at the recipient side.
  - Only among ***DataWriter*** objects attached to the same ***Publisher*** and retrieved among ***DataReader*** objects attached to the same ***Subscriber***
- Data types specified in OMG IDL
  - “Compiled” into type-specific DataReader and DataWriter classes

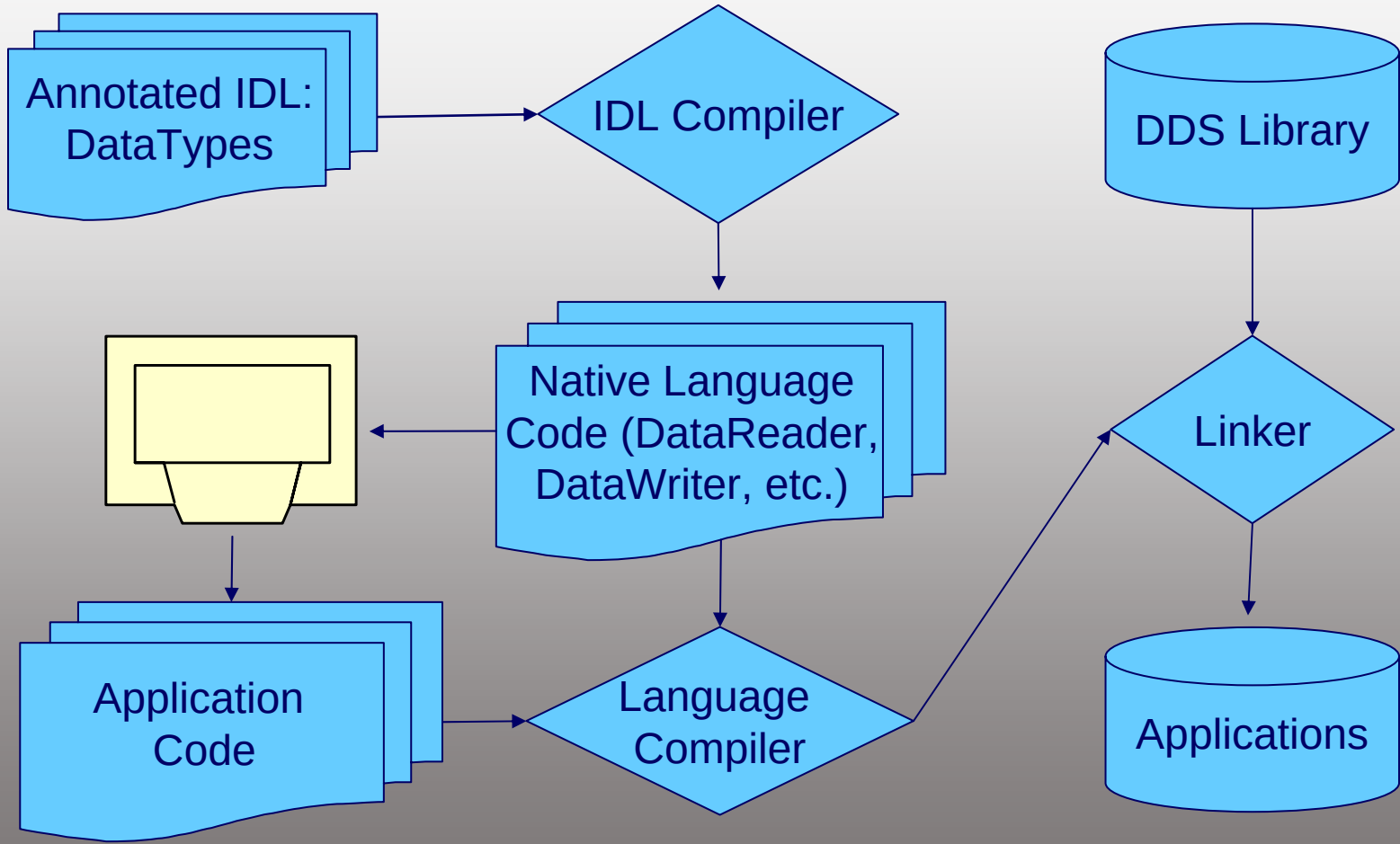
DCPS Data and Data Types
Keys

- May be multiple *instances* of data. Example:
  - Topic = Flight Tracks may contain current data for UA #2333 and US #3456
  - Multiple samples of data for each flight (at different times)
  - Some applications will want to distinguish flights from one another
  - Each data type may have a Key (e.g., Flight Number)
- *Keys* distinguish instances
  - Assumed to be part of data
  - Identified by unspecified method - usually IDL annotation
  - Some QoS settings require use, e.g., those that specify ordering

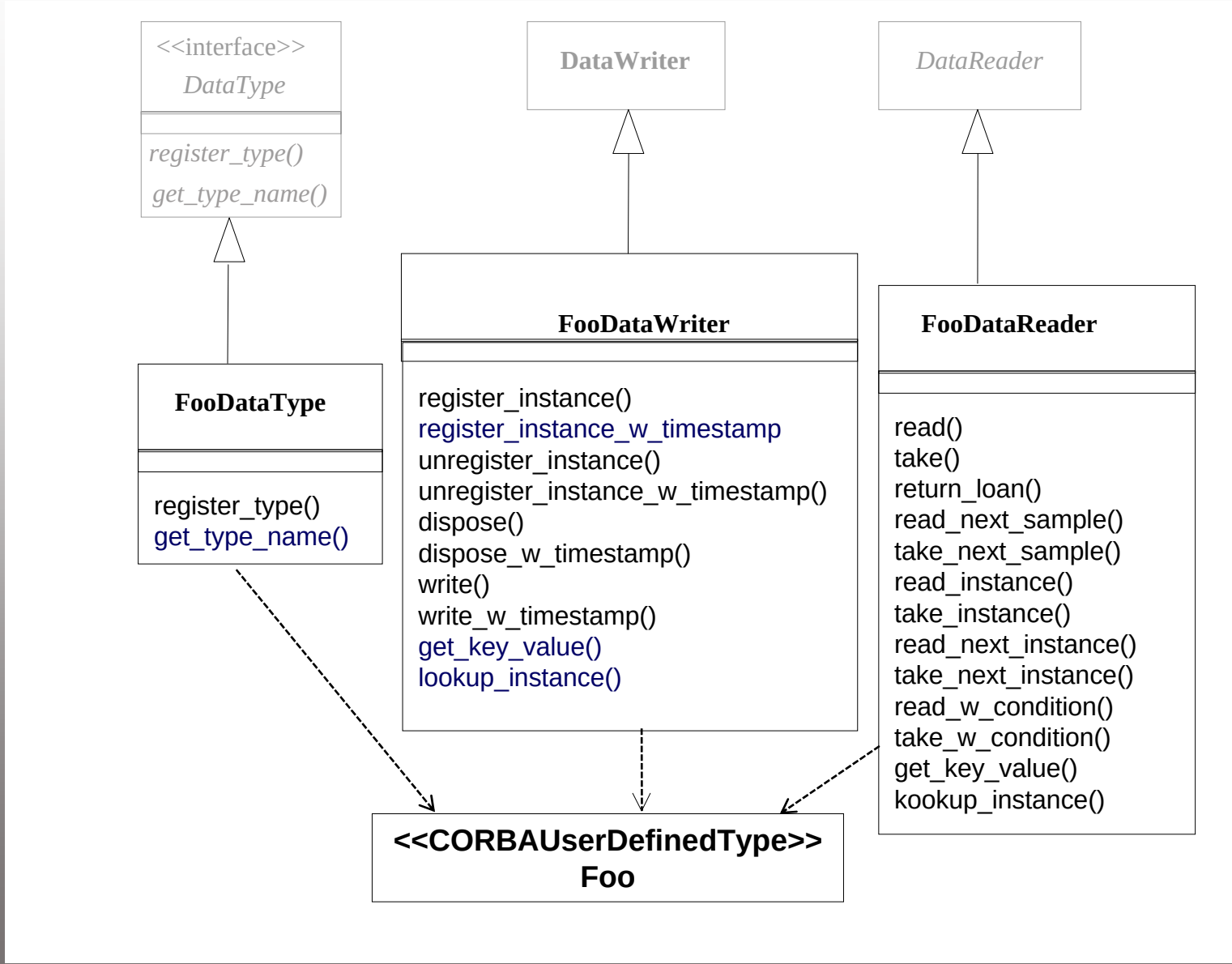
DCPS Data and DataTypes
Why OMG IDL?

- OMG's Interface Definition Language (IDL)
  - Includes a robust, portable type model
  - And constructs for aggregate types, for example:
    - Structures
    - Arrays
    - Sequences
- Part of the CORBA specification
  - Extensively used
  - ISO Standard - ISO/IEC 14750:1999
- A declarative language mapped to modern programming languages
  - Ada 95, C, C++, Java, Smalltalk...

DCPS Data and DataTypes
DDS Development Process (Typical)



DCPS Data and Data Types
Derived Interfaces



DCPS Data and Data Types
Derived Interfaces

- FooDataType
  - `register_type()` – registers the Foo type with the service
  - `get_type_name` – returns “Foo”
- FooDataWriter
  - `register_instance()`
    - Register of an instance of Foo
    - Allows internalization of key value
  - `unregister_instance()` – no longer modifying instance
  - `dispose()` – delete instance; no further publication
  - `write()` – new values for an instance
  - `get_key_value()` – extract key from instance
  - `lookup_instance()`
    - Returns instance handle
    - Instance handle accepted by other operations as optimization
  - *`xxx`*`_w_timestamp()` – specifies timestamp to be used in ordering

DCPS Data and Data Types
Derived Interfaces

OBJECTIVE INTERFACE

- FooDataReader
  - `read()`
    - Read a number of samples consistent with
      - The QoS settings
      - Sequence sizes provided in parameters
    - Copy or no copy possible
      - No copy requires call to `return_loan()`
    - Samples may be “read” again
  - `take()`
    - Read a number of samples as above
    - Samples will not be “read” again by this DataReader
  - `read/take_next_sample()` – next (1) sample
  - `read/take_instance()` – constrained to single instance
  - `read/take_next_instance()` - constrained to next instance
  - *`xxx`*`_w_condition()`- Constrain samples by `ReadCondition`

Quality of Service (QoS)

- QoS values can be associated with most entities
- QoS provides a generic mechanism to control and tailor the behavior of the Service
- Each ***Entity*** supports its own specialized kind of QoS policies
  - QoS policies summarized later
  - The complete list of QoS policies is described in Section 2.1.3 of the specification
- Example: `DESTINATION_ORDER`
  - Controls how each subscriber resolves the final value of a data instance that is written by multiple ***DataWriter*** objects running on different nodes
  - Options: BY_RECEPTION_TIMESTAMP, BY_SOURCE_TIMESTAMP
  - May be applied to: `Topic, DataReader, DataWriter`

QoS Negotiation

- Certain QoS values must be consistent between Publishers and Subscribers. Example:
  - QoS setting DEADLINE specifies rate at which data samples will be provided or are requested
  - Rate offered by publisher must be same or greater than that required by subscriber
- Scenario
  - Subscriber requests a QoS value setting
  - Publisher “offers” a QoS value setting
  - Compatibility determined
    - Compatible – communication between publisher and subscriber
    - Incompatible – signaled to both subscriber and publisher apps
- For each QoS setting, specification contains
  - Need for negotiation
  - Compatibility requirements

QoS Summary

- USER_DATA, GROUP_DATA, TOPIC_DATA
  - Data not interpreted by DDS
  - Distributed as part of the built-in topics
  - User defined extensibility
- DURABILITY – whether data should outlive the source time – VOLATILE, TRANSIENT_LOCAL, TRANSIENT, PERSISTENT
- DURABILITY_SERVICE – configuration if TRANSIENT or PERSISTENT
- PRESENTATION – Scope, coherence, and ordering of data changes

QoS Summary (cont.)

- DEADLINE – periodicity of change
- LATENCY_BUDGET – hint of allowed latency write to read
- OWNERSHIP – allowance for multiple sources: SHARED, EXCLUSIVE
- OWNERSHIP_STRENGTH – determines primary source when EXCLUSIVE
- LIVELINESS – mechanism used by primary to assert “liveness”: duration & AUTOMATIC, MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC
- TIME_BASED_FILTER – minimum separation between deliveries to a DataReader

QoS Summary (cont.)

- PARTITION – “subdomain”
- RELIABILITY – RELIABLE, BEST_EFFORT & max blocking time (for writes)
- TRANSPORT_PRIORITY – hint to infrastructure
- LIFESPAN – “expiration time” for data written
- DESTINATION_ORDER – order of delivery to reader: BY_RECEPTION_TIMESTAMP, BY_SOURCE_TIMESTAMP

QoS Summary (cont.)

- HISTORY – control consolidation of undelivered samples: KEEP_ALL, KEEP_LAST & depth
- RESOURCE_LIMITS – max_samples, max_instances, max_sample_per_instance
- ENTITY_FACTORY – enable each entity implicitly
- WRITER_DATA_CYCLE – auto-dispose instances when unregistered?
- READER_DATA_CYCLE – “autopurge_nowriter_samples_delay” & “autopurge_disposed_samples_delay”

Listeners

- A listener can be attached to each entity
  - Each entity has a specific listener type, e.g., DataReaderListener for a DataReader
- Listener is alerted to relevant state changes in the entity. E.g, DataReaderListener is notified
  - on_requested_deadline_missed()
  - on_requested_incompatible_qos()
  - on_sample_rejected()
  - on_liveliness_changed()
  - on_data_available()
  - on_subscription_match()
  - on_sample_lost()
- Listeners may query status and control behavior of entity in notification.

Conditions and WaitSets

- Condition
  - Base class for triggerable conditions
  - GuardCondition – trigger under control of application
  - StatusCondition
    - Trigger on masked status of any entity
    - Retrieved from any entity by `get_statuscondition()`
  - ReadCondition – allow an application to specify the data samples it is interested in by specifying the desired
    - sample-states
    - view-states
    - and instance-states
- WaitSet - allows an application to wait
  - until one or more of the attached Condition objects has a trigger_value of TRUE
  - or until the timeout expires.

Examples

- C++ code used for illustration
- Scenarios following
  - Bootstrap/Startup
  - Publishing data
  - Subscribing to a Topic
  - Reading with a Listener
  - Wait-based Reading

Bootstrap/Startup: Steps

1. Obtain DomainParticipantFactory
2. Specify Quality of Service for DomainParticipant
3. Optionally, create a DomainParticipantListener and mask of events that listener will react to
4. Join a Domain by creating DomainParticipant

Example – Bootstrap/Startup

OBJECTIVE INTERFACE

```
DomainParticipantFactory domain_factory
    = DomainParticipantFactory.get_instance();

DomainParticipantQoS domain_qos;

DomainParticipantListener*
    domain_participant_listener = 0;

StatusMask domain_participant_mask = 0;

DomainParticipant domain
    = domain_factory->create_participant(
          (DomainId_t) 1,    /* domain_id */
          domain_qos,
          domain_participant_listener,
          domain_participant_mask);
```

Publishing Data: Steps

1. Create Publisher object with appropriate QoS and optional listener
2. Create Topic object with proper name supporting a data type with appropriate QoS and optional listener
3. Create DataWriter object for topic with appropriate QoS and optional listener
4. Declare an instance of the data type
5. Set the value of the instance (not shown)
6. Publish the instance sample

Example – Publishing Data

```
Publisher publisher = domain->create_publisher(
       publisher_qos,
       publisher_listener,
       publisher_listener_mask);

Topic topic = domain->create_topic(
       “Track”, “TrackStruct”,
       topic_qos, topic_listener, mask);

DataWriter writer = publisher->create_datawriter(
       topic, writer_qos, writer_listener, mask);

TrackStruct my_track;

writer->write(&my_track, HANDLE_NIL);
```

Subscribing to a Topic: Steps

1. Create Subscriber object with appropriate QoS and optional listener
2. Create Topic object with proper name supporting a data type with appropriate QoS and optional listener
3. Create DataReader object for topic with appropriate QoS and optional listener

Example – Subscribing to a Topic


```
Subscriber subs = domain->create_subscriber(
      subscriber_qos, subscriber_listener, sub_mask);

Topic topic = domain->create_topic(
       “Track”, “TrackStruct”,
       topic_qos, topic_listener, topic_mask);

DataReader reader = subscriber->create_datareader(
       topic, reader_qos, reader_listener, read_mask);

// reading can be listener-based or wait-based
```

Reading with a Listener: Steps

1. Declare class that inherits from DataReaderListener
2. Override on_data_available to
    1. Take available data
    2. Process it
3. Create instance of listener class
4. Set as listener on DataReader (or supply to `create_datareader`)

Example – Reading with a Listener

OBJECTIVE INTERFACE

```
class MyReadListener : DataReaderListener {
    void on_data_available( DataReader reader); }

ReadListener::on_data_available( DataReader reader )
{
    FooSeq received_data(5);
    SampleInfoSeq sample_info(5);

    reader->take( &received_data, &sample_info,
           5, /* max_samples */, …);

    // Use received_data
}

Listener listener = new MyReadListener();
reader->set_listener(listener, listen_mask);
```

Wait-based Reading: Steps

1. Create ReadCondition with proper SampleStateMask, ViewStateMask, and InstanceStateMask
2. Attach condition to WaitSet
3. Wait on WaitSet
4. Declare sequence to receive data and information about samples
5. Take data with condition
6. Process data

Example – Wait-based Reading

OBJECTIVE INTERFACE

```
Condition foo_condition =
   reader->create_readcondition(…);

waitset->attach_condition(foo_condition);

ConditionSeq active_conditions;
waitset->wait(&active_conditions, timeout);

FooSeq received_data(5);
SampleInfoSeq sample_info(5);

reader->take_w_condition(&received_data,
                         &sample_info,
                         5, /* max samples */
                         foo_condition);
// Use received_data
```

Summary

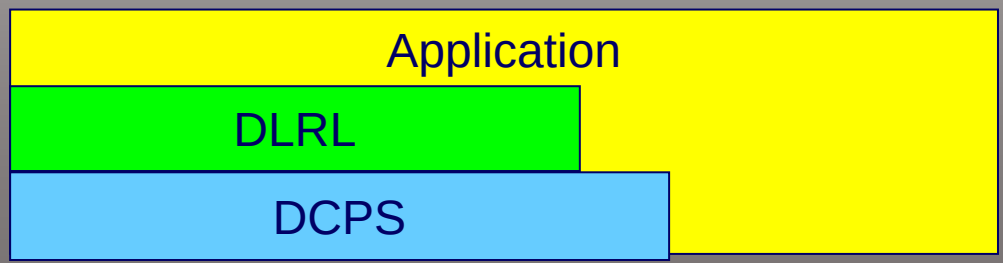
- DDS Data Centric Publish – Subscribe (DCPS) allows
  - Publishing applications to
    - Identify the data objects they intend to publish, and
    - Provide values for these objects.
  - Subscribing applications to
    - Identify which data objects they are interested in, and
    - Access their data values.
  - Applications to
    - Define topics
    - Attach type information to the topics
    - Create publisher and subscriber entities
    - Attach QoS policies to all these entities

Data Local Reconstruction Layer (DLRL)

Part II – Data Distribution Service

Data Local Reconstruction Layer (DLRL)

- Optional higher level
- Automatically reconstructs data locally from distributed updates
- Allows the application to access the data ‘as if’ it were local
- Propagates information to all interested subscribers but also updates a local copy of the information.
- Can be built on top of DCPS



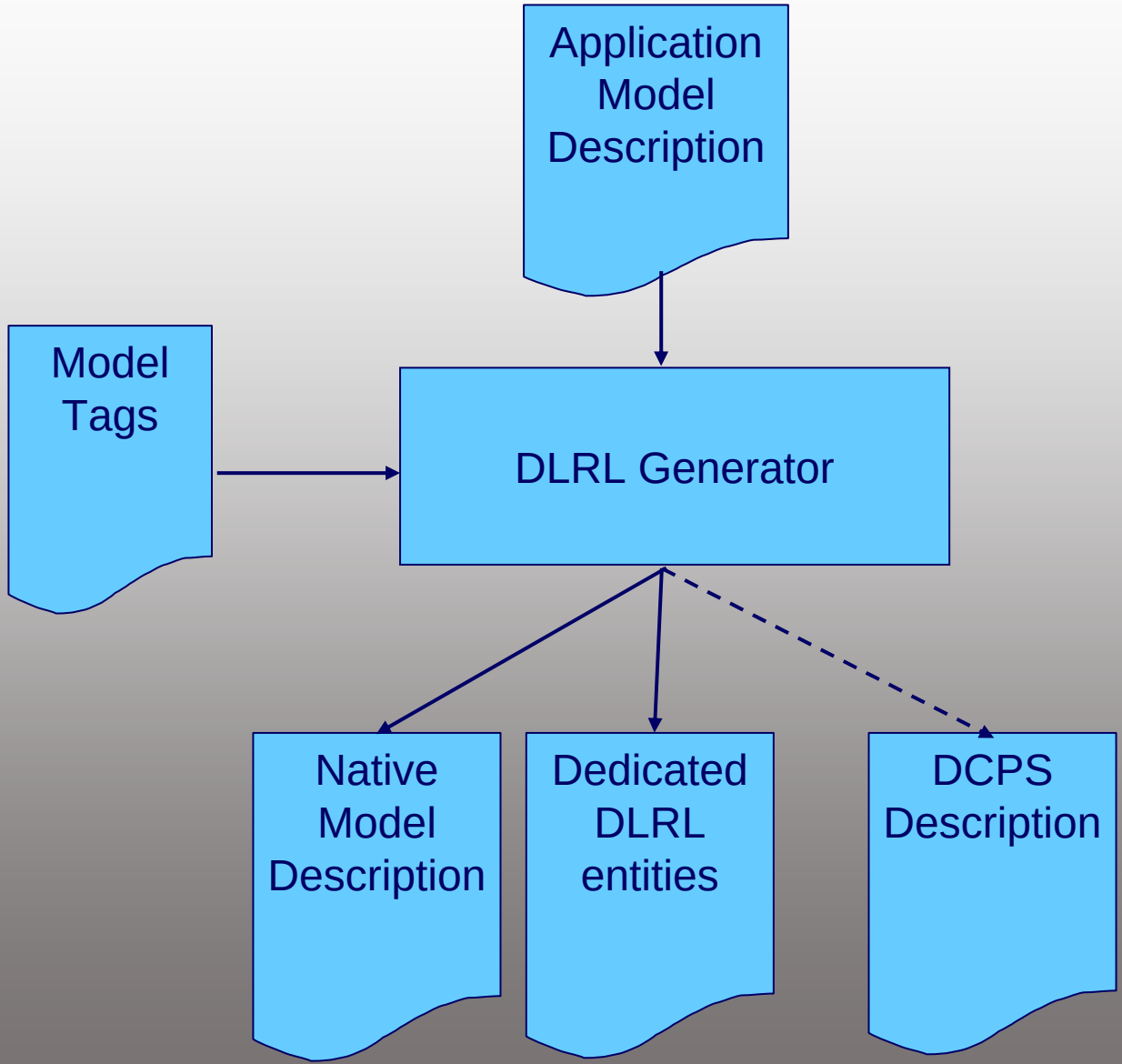
DLRL

- Application developer will be able to
  - Describe classes of application objects with their
    - Methods,
    - Data fields and
    - Relations;
  - Attach some of those data fields (shared) to DCPS entities
  - Manipulate those objects (i.e., create, read, write, delete)
    - Using the native language constructs
    - Activates, behind the scenes, the attached DCPS entities
  - Manage those objects in a cache of objects
    - Ensuring all references that point to a given object actually point to the same language cell
- Applications objects mapped to DCPS entities
  - Designated by a set of annotation tags

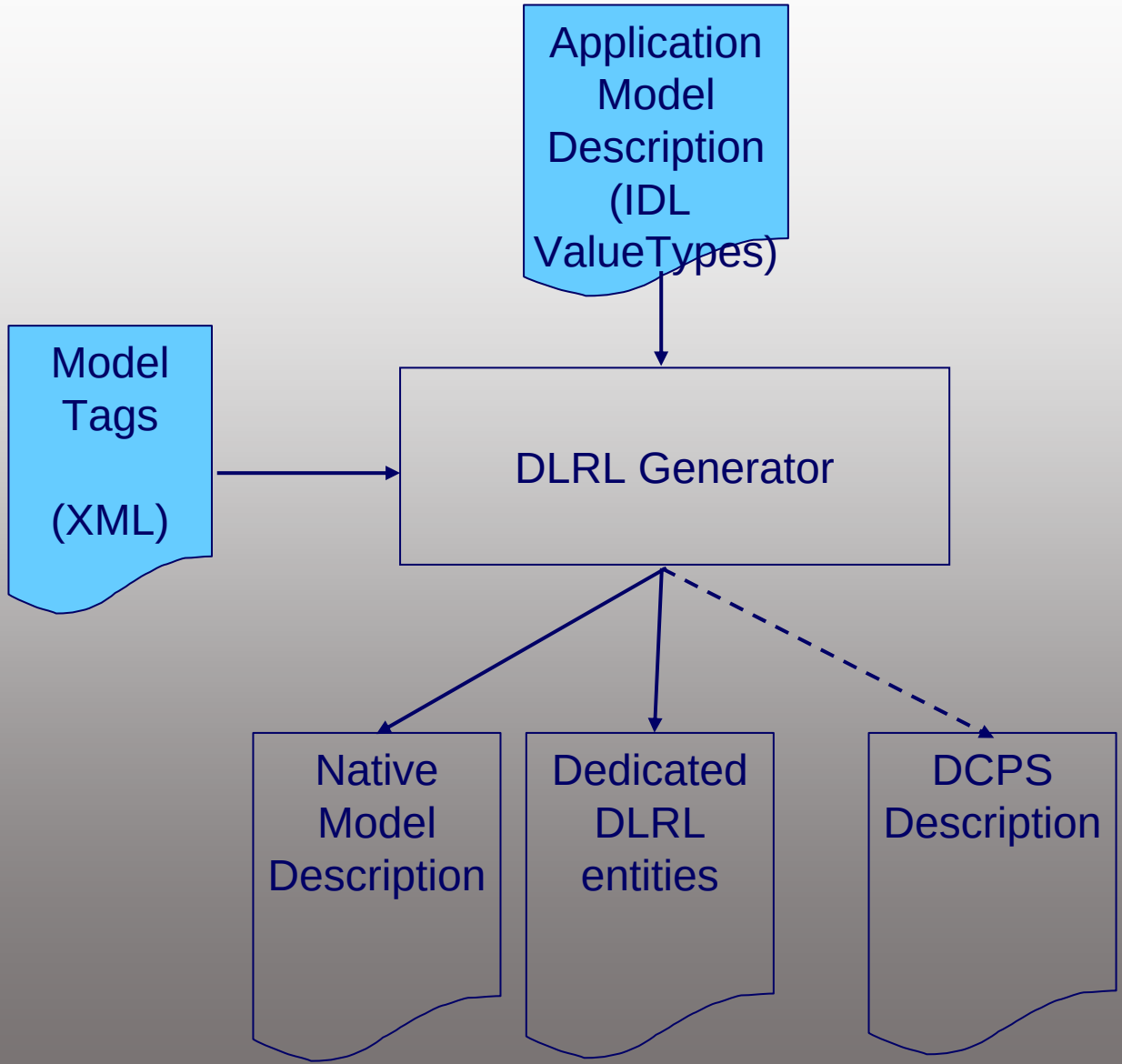
DLRL Generation Process

- Based on the
  - Application model description (IDL ValueTypes)
  - Tags that enhance the description (XML)
- Tool will generate:
  - Native model definition
    - Application classes usable by the application developer
    - IDL
  - Dedicated DLRL entities
    - Helper classes to consistently use the application class
    - Form the DLRL run-time
    - IDL
  - On demand, the corresponding DCPS description
- IDL is compiled into programming language

DLRL Generation Process



DLRL Generation Process



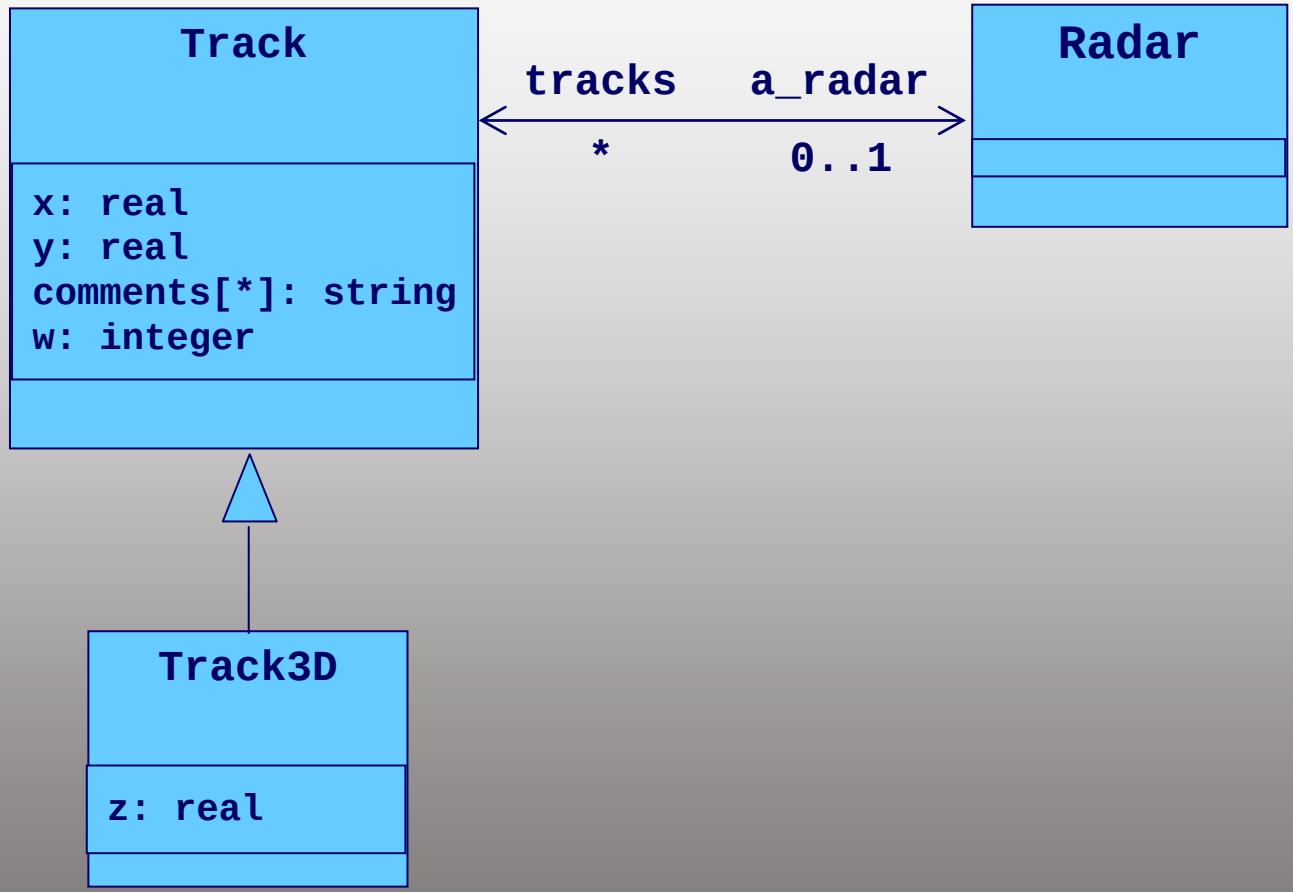
Application Model

- Objects with
  - Methods
  - Attributes which can be
    - Local – do not participate in the data distribution
    - Shared
      - Participate in the data distribution process
      - Attached to DCPS entities
- Related by
  - Inheritance between classes
    - Single inheritance from other DLRL objects
    - Multiple inheritance from native language objects
  - Associations between instances
    - Use-relations (no impact on the object life-cycle)
    - Compositions (constituent object lifecycle coincides with the compound object's one)

Application Model

- Shared attributes can be
  - Mono-valued:
    - Of a simple type:
      - basic-type (long, short, char, string, etc.);
      - enumeration-type;
      - simple structure
    - Reference to a DLRL object.
  - Multi-valued (collections of homogeneously-typed items)
    - List (ordered with index)
    - Map (access by key)

Example Application Model - UML



Example Application Model - IDL

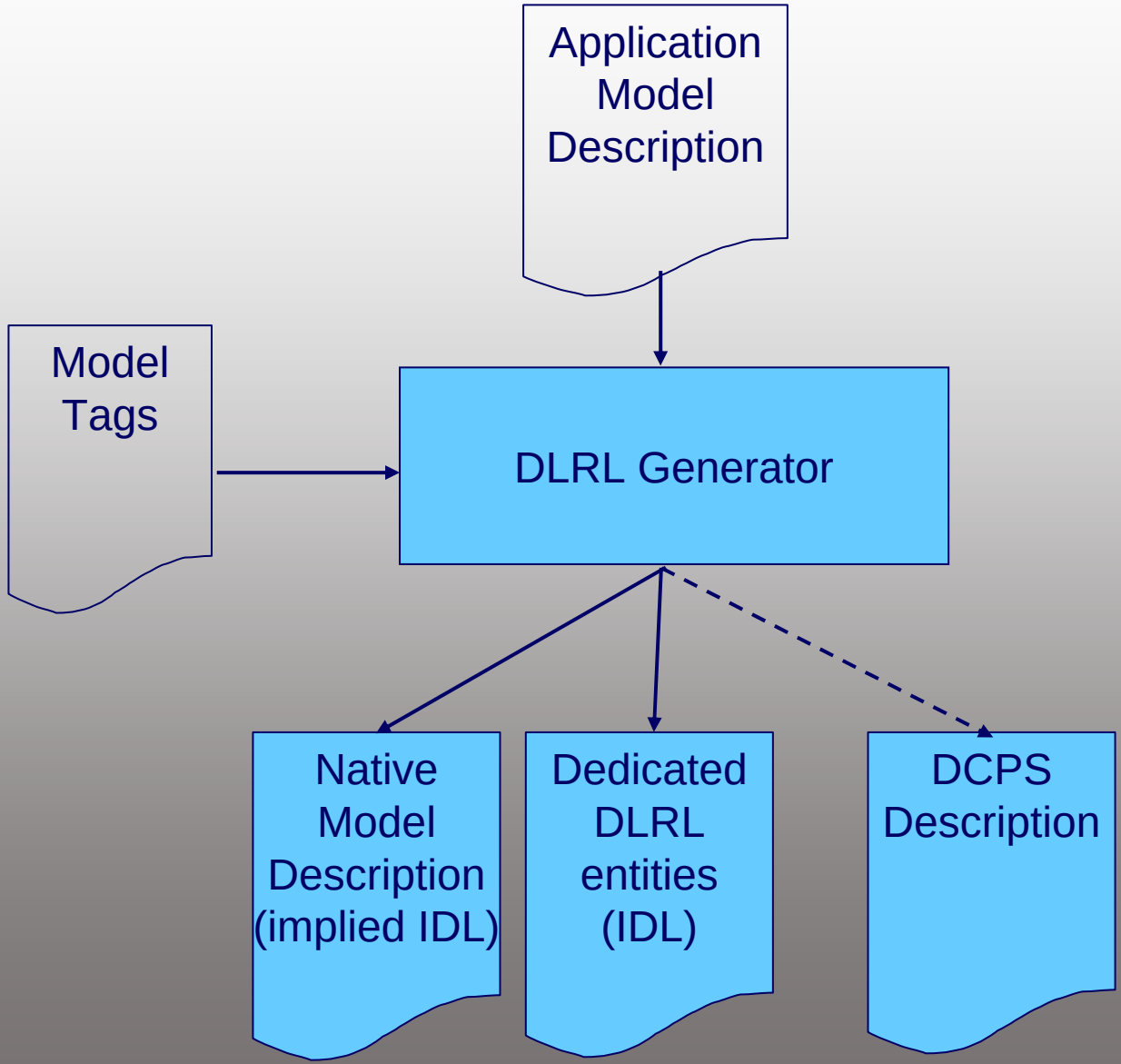
```
#include "dlrl.idl"

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;
    public RadarRef     a_radar;
    };
valuetype Track3D : Track {
    public double z;
    };
valuetype Radar : DLRL::ObjectRoot {
    public TrackList tracks;
    };
```

Model Tags

- Symbolic representation of IDL
- Mapping to DCPS topics, etc.
- Excerpt:

```
<classMapping name="Track">
    <mainTopic name="TRACK-TOPIC">
        <keyDescription content="FullOid">
            <keyField>CLASS</keyField>
            <keyField>OID</keyField>
        </keyDescription>
    </mainTopic>
<monoAttribute name="x">
    <valueField>X</valueField>
</monoAttribute>
```

DLRL Generation Process



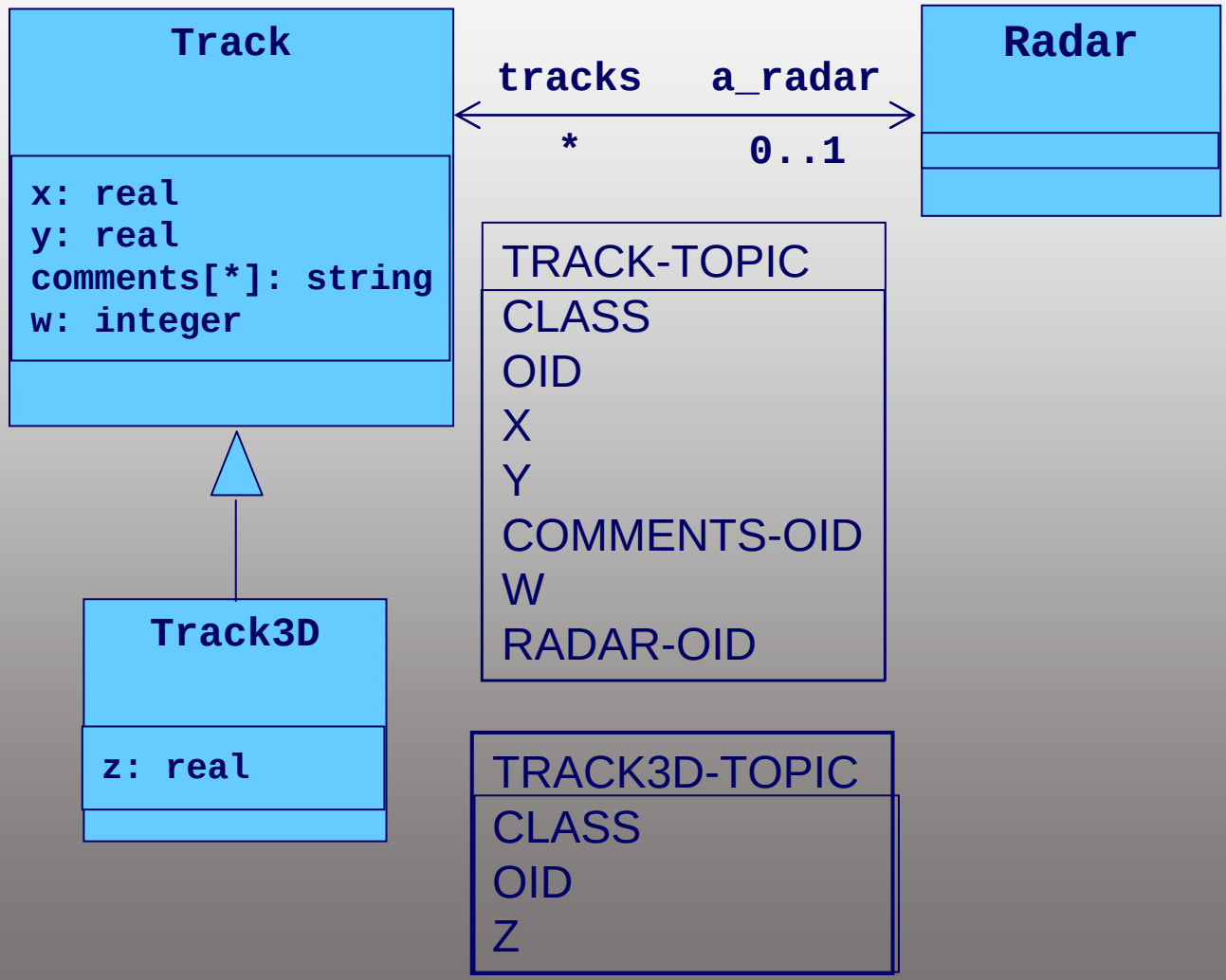
DLRL Generator – Three Mappings

- Structural mapping
  - Relations between DLRL objects and DCPS data;
- Operational mapping
  - Mapping of the DLRL objects to the DCPS entities (Publisher, DataWriter, etc.) including
    - QoS settings
    - Combined subscriptions
    - Etc.;
- Functional mapping
  - Relations between the DLRL functions
    - Mainly access to the DLRL objects
  - and the DCPS functions – write/publish/etc.

Structural Mapping

- Each instance of a DLRL object is mapped to a row (sample) of a DCPS topic
- Data type for topic includes
  - String field for DLRL type name
  - Oid – (integer) identifier of instance
  - Fields mapped from DLRL attributes
    - Simple attributes mapped to field
    - Object references (associations) mapped to Oid, TypeName
    - Collections mapped to row for each element
      - Same key field
      - Distinguished by index in collection

Example Structural Mapping



Operational Mapping

- Each DLRL class is associated with several DCPS Topics
  - Each has a DCPS DataWriter and/or a DCPS DataReader
  - A DataReader and DataWriter are attached to each Cache instance
- All DataWriter and DataReader objects used by a DLRL object are attached to a single Publisher/Subscriber
  - in order to consistently manage the object contents
- Operations are provided at the DLRL level to
  - Create and activate all the DCPS entities
  - Set the QoS of each

Functional Mapping – DLRL entities

- CacheFactory – creates Cache instances
- Cache – a set of DLRL objects that are locally available
- CacheAccess – mediates access to a cache
- ObjectHome – act as representative for all the local instances of a given application-defined class.
- ObjectListener – implemented by the application to be made aware of updates on objects belonging to an ObjectHome.
- Selection – act as representatives of a subset of objects defined by an expression attached to the selection
- ObjectFilter – act as filter for Selection object

Functional Mapping – DLRL entities

- ObjectQuery – specialization of ObjectFilter that performs a filter based on a query expression
- SelectionListener – implemented by the application to be made aware on updates made on objects belonging to that selection.
- ObjectModifier – represent modifiers to be applied to a set of objects.
- ObjectExtent – manages a set of instances
- ObjectRoot – abstract root class for all the application-defined classes.
- ObjectReference – a raw reference (untyped) to an object.
- Reference – a typed reference to an object.

Example: Using CacheAccess for Read

1. Create the ***CacheAccess*** for read purpose (***Cache::create_access***)
2. Clone some objects in it (***ObjectRoot::clone*** or ***clone_object***);
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;
6. Eventually, delete the ***CacheAccess*** (***Cache::delete_access***).

Example: Using CacheAccess for Write

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

Summary

- DDS-DLRL is a layer on top of DCPS to
  - Ease the management of the data
  - Integrate into the application
- Supports
  - Object-orientation
  - Management of graphs of objects
- It promotes typed interfaces
  - By means of code-generation
- It does not force a global object model
  - The object model is local
  - Mapping to the DCPS data model

Compliance Profiles

Compliance Profiles

- *Minimum profile*: mandatory features of the DCPS layer
- *Content-subscription profile*:
  - Adds the optional classes: ***ContentFilteredTopic***, ***QueryCondition***, ***MultiTopic***
  - Enables subscriptions by content
- *Persistence profile*: adds
  - Optional Qos policy DURABILITY_SERVICE
  - Optional settings ‘TRANSIENT’ and ‘PERSISTENT’ of the DURABILITY QoS policy ***kind***
  - Enables saving data into either TRANSIENT memory, or permanent storage so that it can survive the lifecycle of the ***DataWriter*** and system outings.

Compliance Profiles

- *Ownership profile*: adds
  - Optional setting ‘EXCLUSIVE’ of the OWNERSHIP ***kind***
  - Support for the optional OWNERSHIP_STRENGTH policy
  - Ability to set a ***depth*** > 1 for the HISTORY QoS policy.
- *Object model profile*: includes
  - DLRL
  - Support for the PRESENTATION ***access_scope*** setting of ‘GROUP’

Further Information

- Current Specification –
  - http://www.omg.org/cgi-bin/doc?ptc/2005-03-09