
**Angelo Corsaro, Ph.D.**
**Product Strategy & Marketing Manager**
OMG RTESS and DDS SIG Co-Chair
angelo.corsaro@prismtech.com

PRISMTECH
Powering Netcentricity

# Hibernating DDS

## with Java and C++

# Agenda

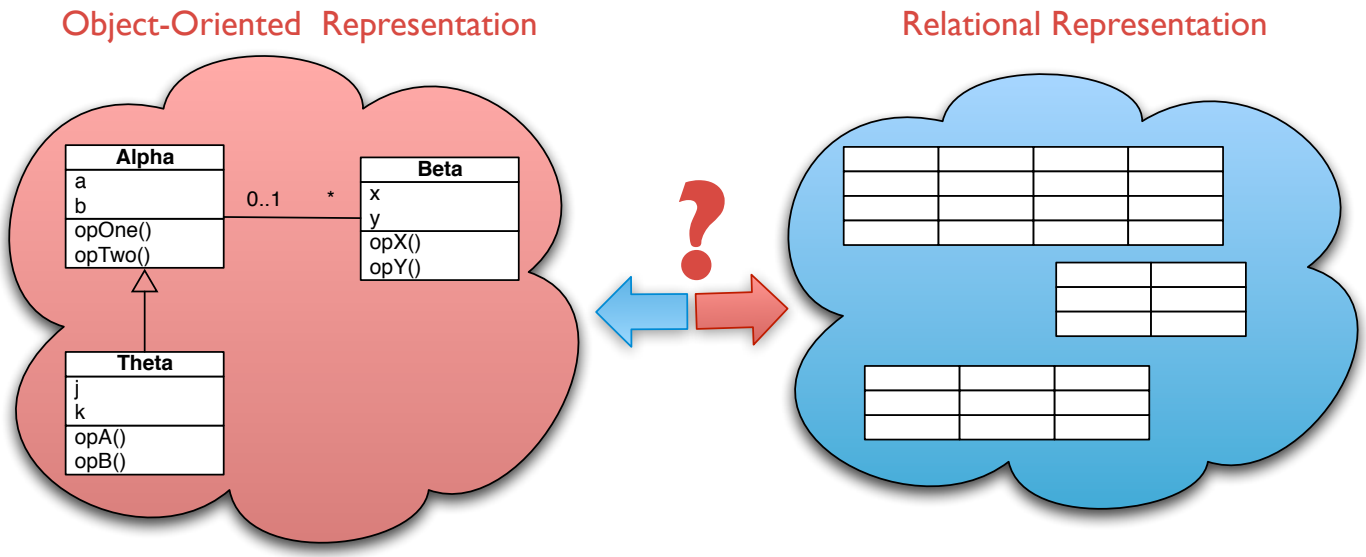
- The Object/Relational Impedance Mismatch
- Why Hibernating DDS?
- DLRL: The Mysterious Acronym
- Hibernating DDS with DLRL
- Concluding Remarks

# Object/Relational Impedance Mismatch [1/2]

- When designing and building application with Object Oriented methodologies and programming languages, one often struggles with the **Object/Relational Impedance Mismatch**
- The Object/Relational Impedance Mismatch arises all the time when data represented in a relational form, e.g., stored in a DBMS, has to be manipulated by applications written in Object Oriented Programming Languages
- In these cases, the solution is often to manually reconstruct applications' objects, and their relationships out of the data base tables
- This approach however, has several drawbacks, as for instance it is tedious, error-prone, requires application programmers to be familiar with SQL, etc.

# Object/Relational Impedance Mismatch [2/2]

- Several approaches have been suggested to gap the object/ relational impedance mismatch
- For the specific case of Data Bases, two approaches have received most attention:
  - Use of OODMBS
  - Use of ORM technologies, such as **JDO**, **Hibernate**, etc.
- However, due to the ubiquitous presence of RDBMS, ORM techniques are currently predominant

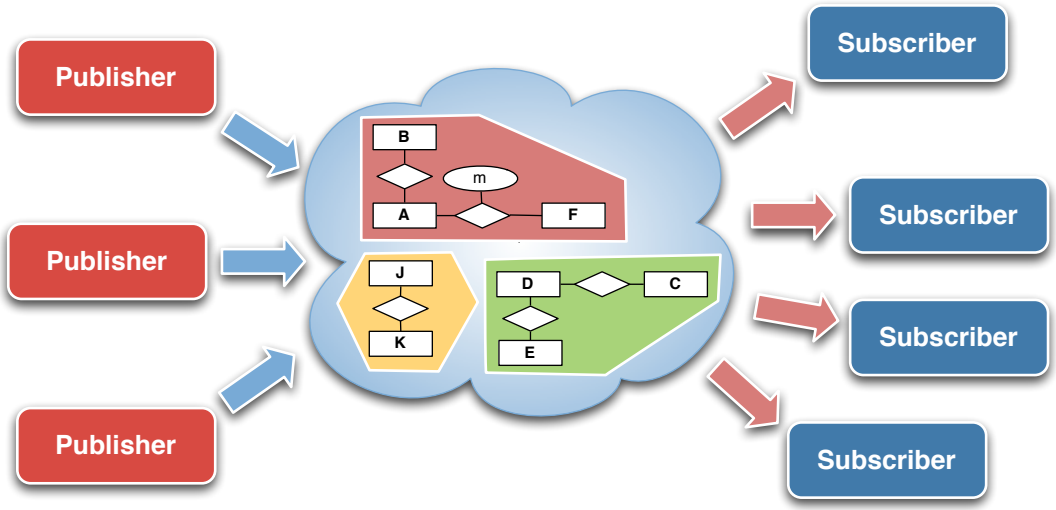


# Agenda

- The Object/Relational Impedance Mismatch
- Why Hibernating DDS?
- DLRL: The Mysterious Acronym
- Hibernating DDS with DLRL
- Concluding Remarks

# Why Hibernating DDS?

- DDS allows application to do publish/subscribe over a **Distributed Relational Data Model**
- DDS Topics, the unit of information that applications can publish/subscribe, can be seen and treated locally, as if it was a DBMS Table
- Thus, each DDS application can use a subset of SQL92 to access and navigate the set of topics it's currently subscribing

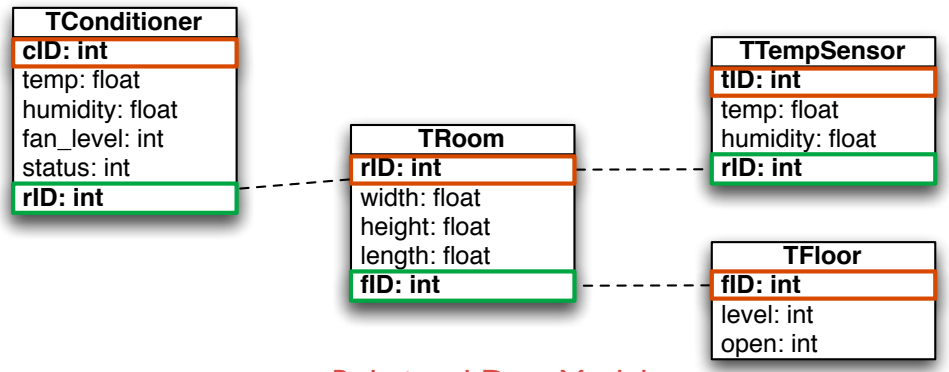



As a result, when using DDS with Object-Oriented Programming Languages the Object/Relational Mismatch needs to be addressed!

# A Simple Example[1/3]

- Suppose we have to build an application for controlling the temperature in big buildings
- The first step is to identify the DDS (actually DCPS) data model capturing the key entities of this application
- Once this is done, we will write the application by leveraging on DDS Readers/Writers and building the business logic on-top of these abstractions

- Topic **Keys** can be used to identify **instances** as well as **relationships**
- **Relationships** can be **navigated** by relying on a subset of **SQL 92**
- **Keys** can be represented by an **arbitrary number of Topic fields**




Relational Data Model

# A Simple Example [2/3]

```
class Room {
public:
    explicit Room(int id);
    Room(int it, float width, float height, float length);
    ~Room();
public:

    void set_climate(const Climate_t c);
    void set_temperature(float temp);
    float get_temperature();
    void set_humidity(float hum);
    float get_humidity();
public:
    // Performs required DDS pub/sub activities
    void update();
    void flush();

protected:
    void init(int id);

private:
    TRoom               room_topic_;
    TRoomReader*        room_reader_;
    TRoomWriter*        room_writer_;

    std::vector<Conditioner> cond_vec_;
    std::vector<TempSensor>  temp_vec_;

    // Used to build and manage the association between the
    // room and the TempSensor
    TTempSensorReader*   temp_reader_;
};
```

```
class Conditioner {
public:
    explicit Conditioner(int id);
    Conditioner(const TConditioner& tcond)
    ~Conditioner();
public:
    void set_temp(float temp);
    void set_humidity(float hum);
    void set_fan_level(int level);
    void start();
    void stop();
    void pause();
public:
    // Performs required DDS pub/sub activities
    void update();
    void flush();
private:
    TConditioner        cond_topic_;
    TConditionerWriter* cond_writer_;
};
```

```
void
Room::init(int id) {
    TTempSensorReader*   temp_reader_ = ...; //Resolve the reader
    TConditionerReader*  cond_reader_ = ...; //Resolve the reader

    // Query the reader for the TTempSensor and TConditioner
    // that have "rID == id" then add those to the container

}
```

# A Simple Example [3/3]

- As the simple example reveals, if DDS (actually DCPS) is used from an Object Oriented programming language such as Java, or C++, the programmer is either left to deal with the Object/Relational Mismatch
- Depending on how the programmer tackles this mismatch, its resulting application will be more or less polluted by "DDS" details
- Implementing by hand the Object/Relational Mapping is quite a bit of work, which becomes quickly non-creative and repetitive

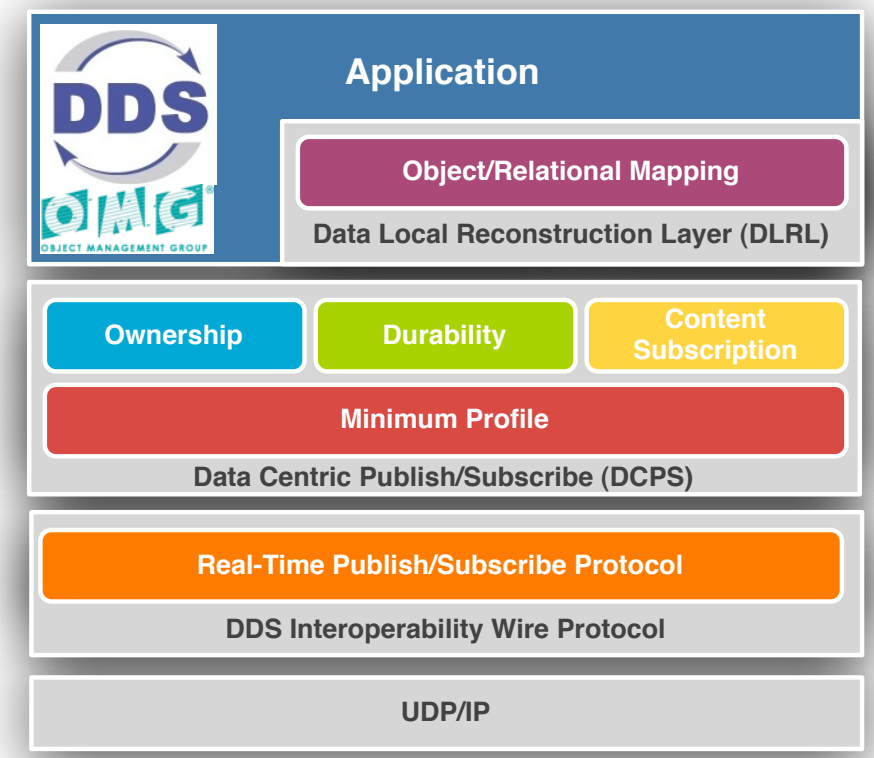
**Is there a better way to take tackle the Object/Relational Impedance Mismatch in DDS?**

# Agenda

- The Object/Relational Impedance Mismatch
- Why Hibernating DDS?
- **DLRL: The Mysterious Acronym**
- Hibernating DDS with DLRL
- Concluding Remarks

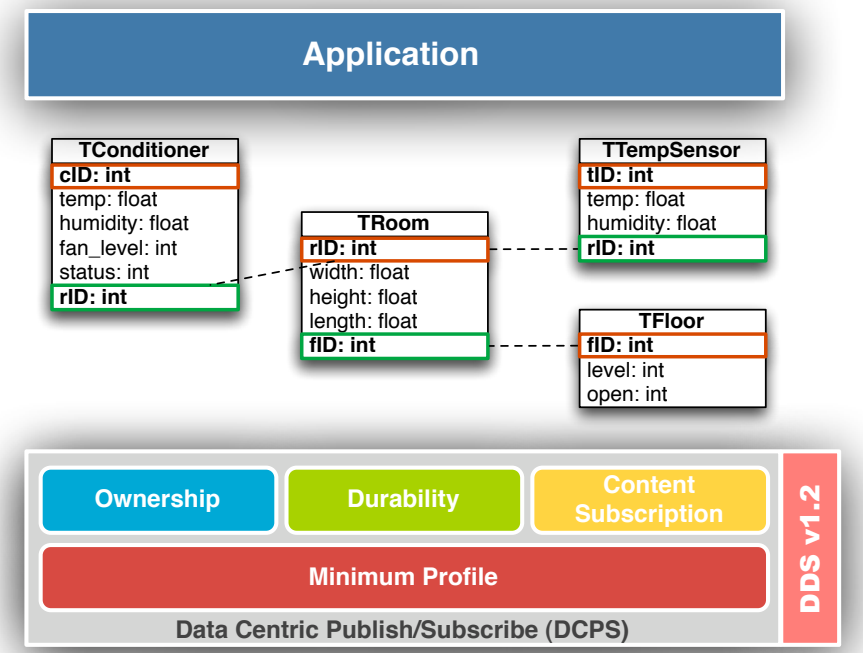
# DLRL: The Mysterious Acronym…

- The OMG DDS Standard included since its inception an optional layer, namely the **Data Local Reconstruction Layer (DLRL)**, which provided an extended Object/Relational Mapping facility for DDS
- The DLRL is quite similar to Hibernate (or Java Data Objects), although introduces some extensions needed to deal with Distributed Systems
- DLRL implementation currently exist for C++ and Java



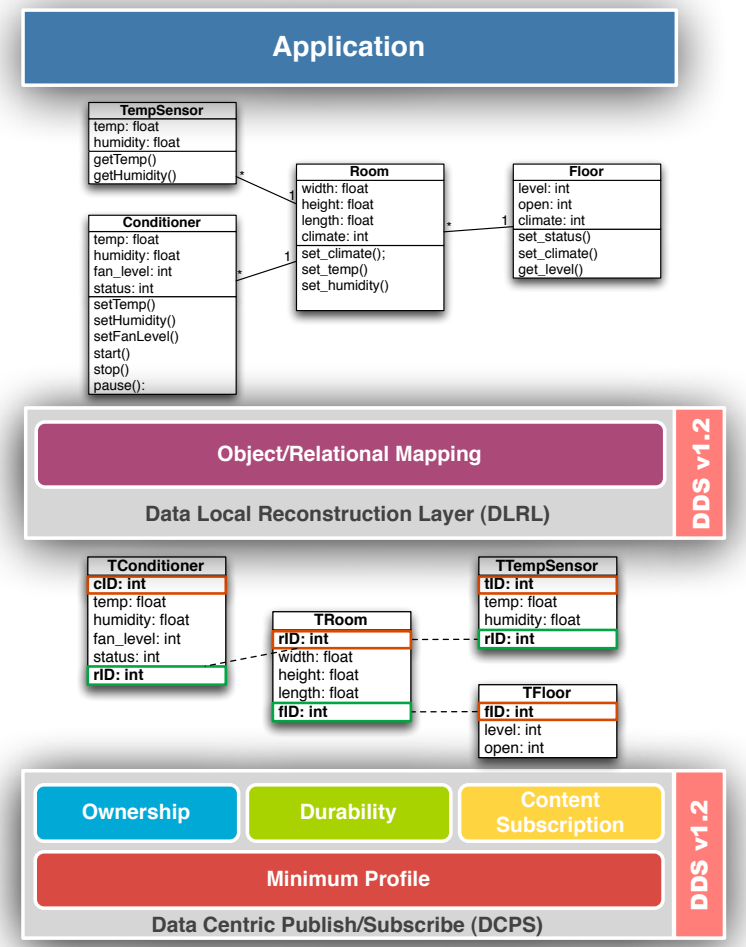
# DCPS Application

- The application works directly at the DCPS level
- When using OO Programming Languages the Object/Relational Impedance Mismatch has to be manually dealt



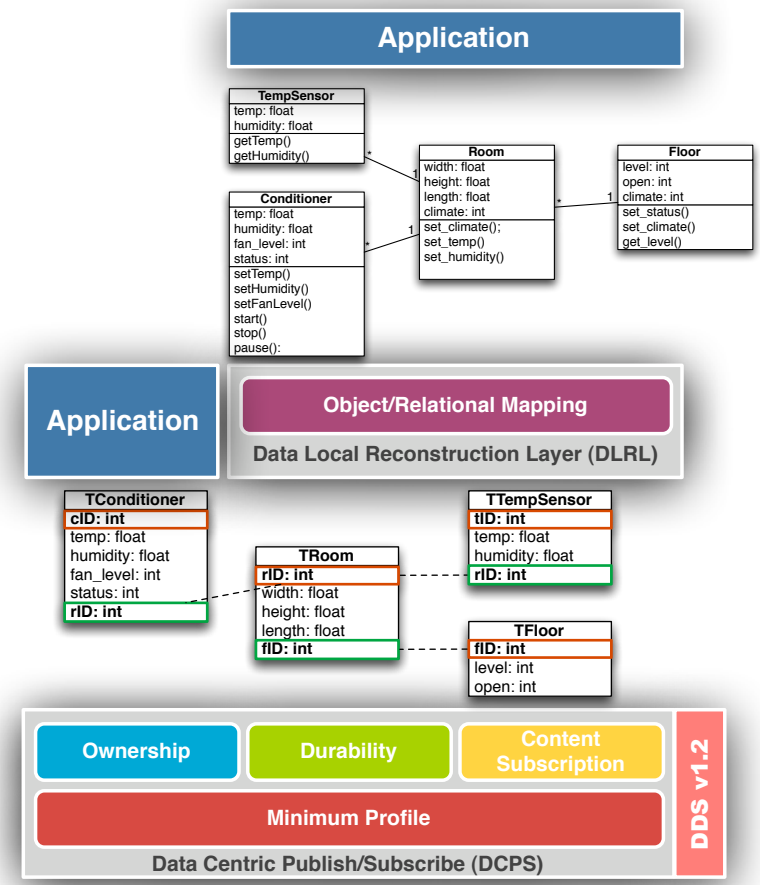
# DLRL Application

- The DLRL Layer is used to provide a Language Integrated access to DDS data
- The Designer has great freedom in deciding how Objects have to map to Topics
- Different Object Reconstruction can be created for different applications



# Generic DDS Application

- In the most general case, different portion of the application might rely on DLRL or DCPS depending on their specific needs
- DCPS access might be required for accessing and tuning some specific QoS



# Agenda

- The Object/Relational Impedance Mismatch
- Why Hibernating DDS?
- DLRL: The Mysterious Acronym
- **Hibernating DDS with DLRL**
- Concluding Remarks

# DLRL Key Features

- The main goal of DLRL is that of providing a seamless integration of DDS for Object Oriented Programming Languages
- DLRL provides a seamless integration for both:
  - Applications that want to completely ignore the relational nature of DCPS
  - Application that need to leverage an existing DCPS Data Model
- DLRL provides support for:
  - Local/Shared Attributes (only shared attributes are subject to data distribution)
  - Mono-/Multi-valued attributes
  - Association, with support for both one-to-one, one-to-many
  - Aggregation, with support for both one-to-one, one-to-many
  - Single Inheritance

# Structural Mapping

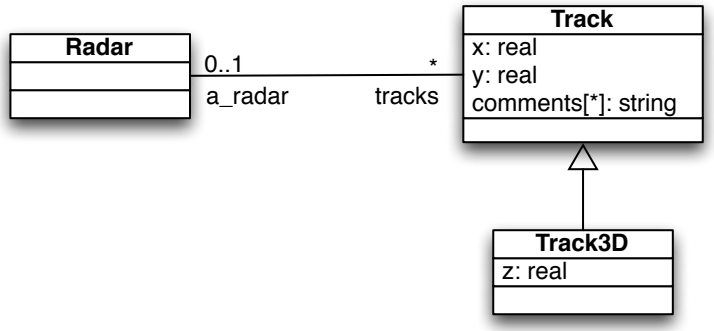
**Two approaches are possible for structural mapping of DLRL Objects to DCPS**

1. DLRL mapping to DCPS Topics can be automatically inferred by the middleware using default mapping rules

2. DLRL mapping to DCPS Topics can be specified by the designer, thus making it possible for reconstructing Object Oriented views of existing data models

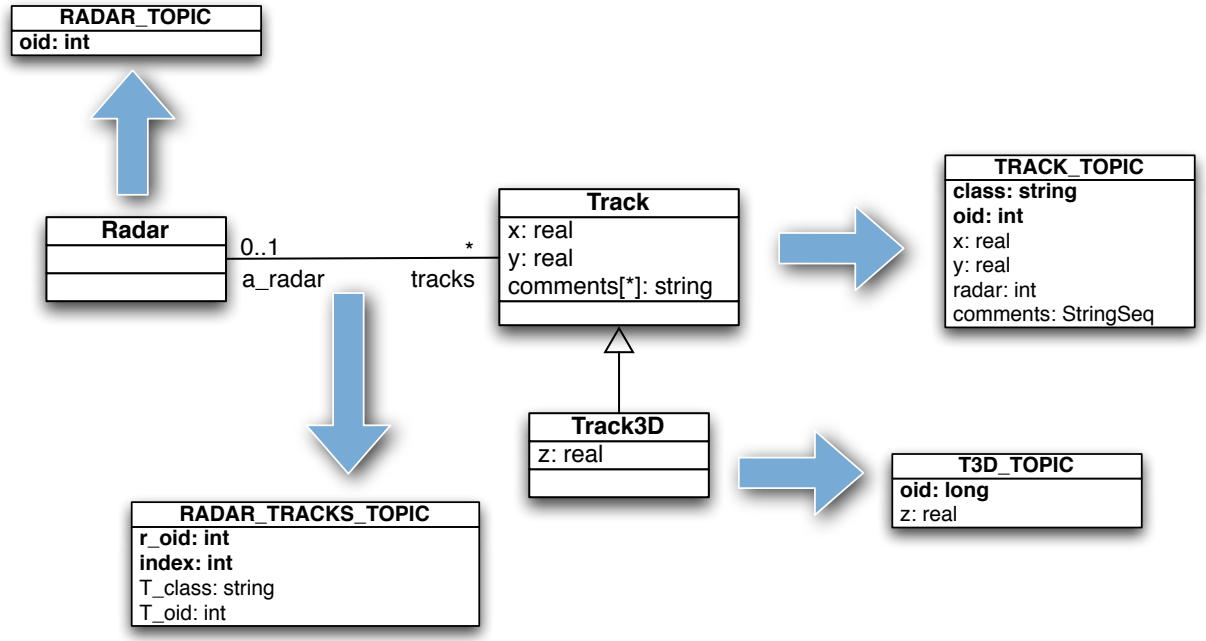
# Structural Mapping

1. DLRL mapping to DCPS Topics can be automatically inferred by the middleware using default mapping rules



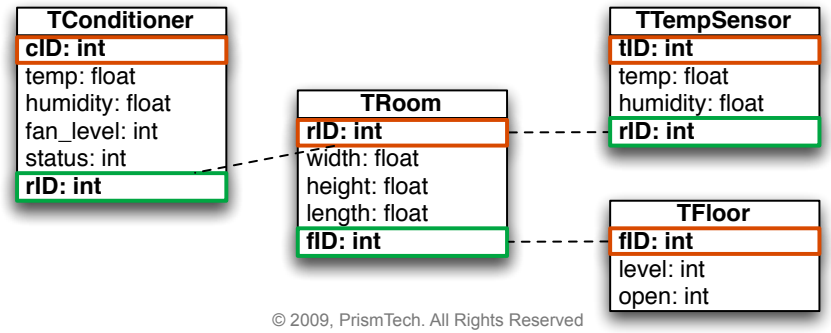
# Structural Mapping

1 DLRL mapping to DCPS Topics can be automatically inferred by the middleware using default mapping rules



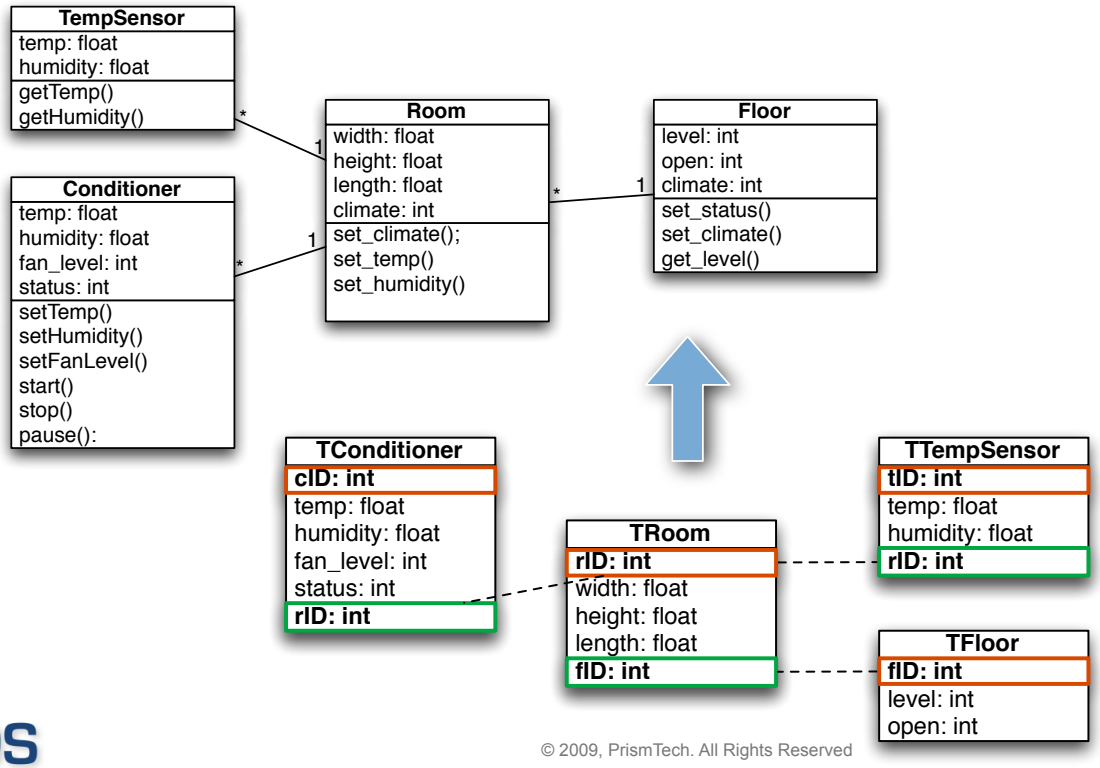
# Structural Mapping

2 DLRL mapping to DCPS Topics can be specified by the designer, thus making it possible for reconstructing Object Oriented views of existing data models



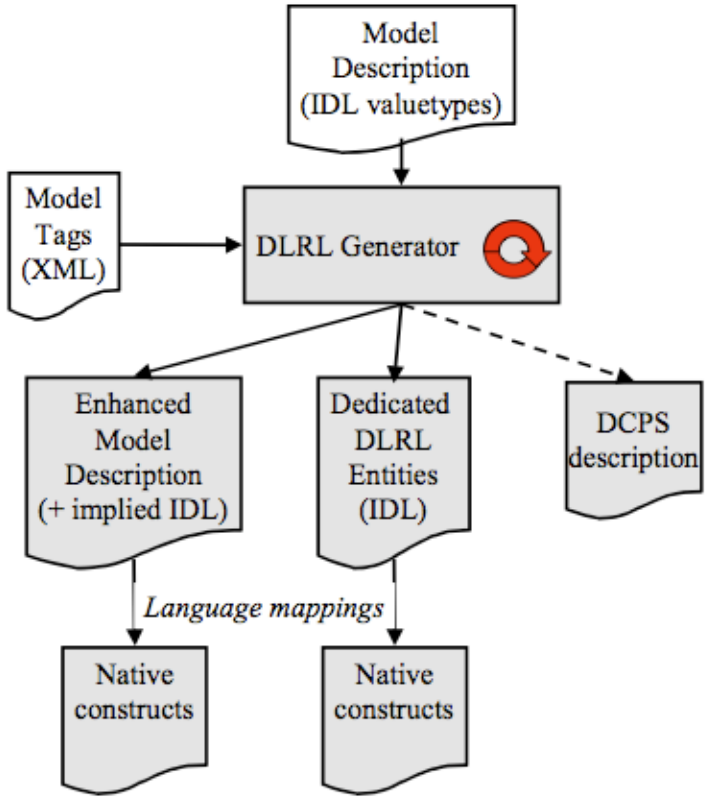
# Structural Mapping

2 DLRL mapping to DCPS Topics can be specified by the designer, thus making it possible for reconstructing Object Oriented views of existing data models

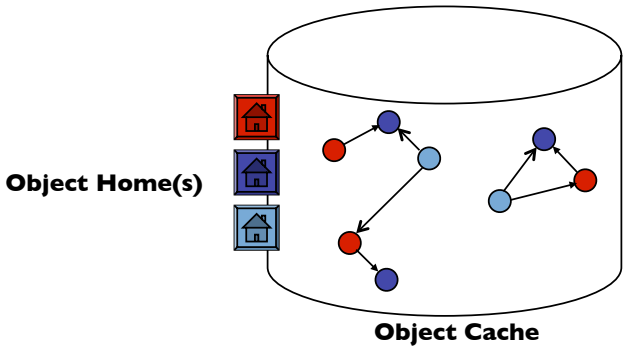


# Specifying the Structural Mapping

- The structural mapping between DLRL and DCPS entities is specified by relying, depending on the case, on:
  - The IDL description of the DLRL Objects
  - The Model Tags defining the mapping (which could be default)
  - An Optional IDL file specifying DCPS Topics



# Working with DLRL Objects

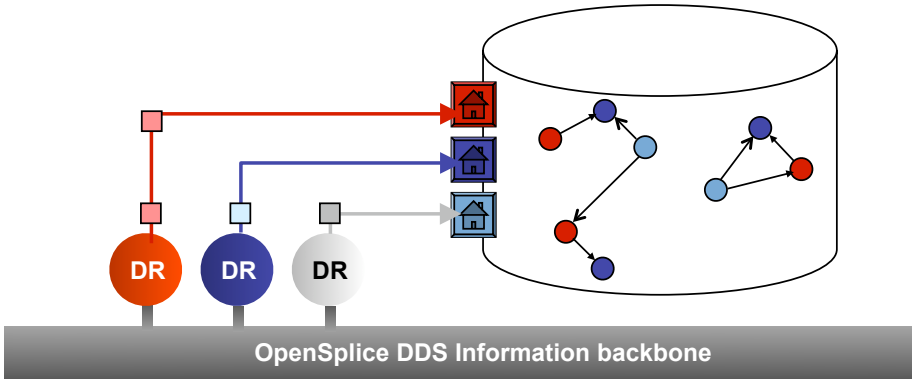



## Concepts

The mechanism at the foundation is a managed Object Cache:

- An Object Cache can be populated by different types (classes) of Objects.
- Each object class has its own manager called an ObjectHome.
  - They can inform the application about object creation/modification/deletion.
- Classes may contain navigable relationships to other classes.
- Each Object class may inherit from 1 other Object class.

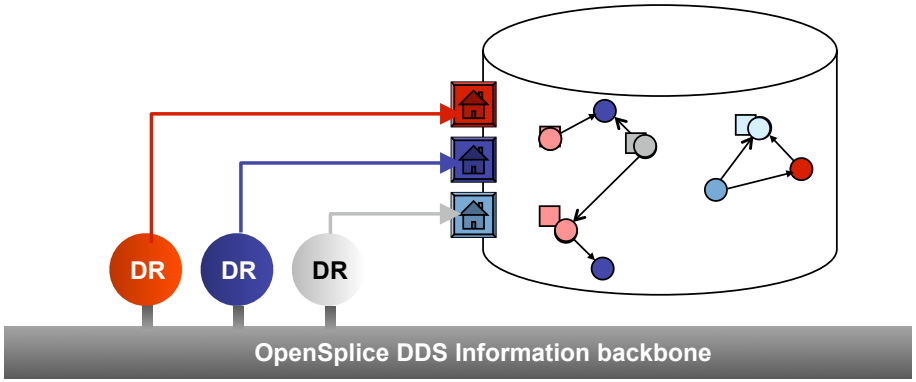
# Working with DLRL Objects




## Processing Updates

- DCPS updates arrive as separate samples at separate times.
- DLRL Updates are processed in 'update rounds':
  - ObjectHomes read all available samples from the DDS information backbone and update their corresponding objects in the Cache accordingly.
  - Objects are allocated once and their state is 'overwritten' on subsequent updates.
  - Therefore an Object always contains the latest available state.
  - Push mode: update rounds start when new data arrives. The application gets notified by Listeners.
  - Pull mode: the application can determine the start of each update round manually.

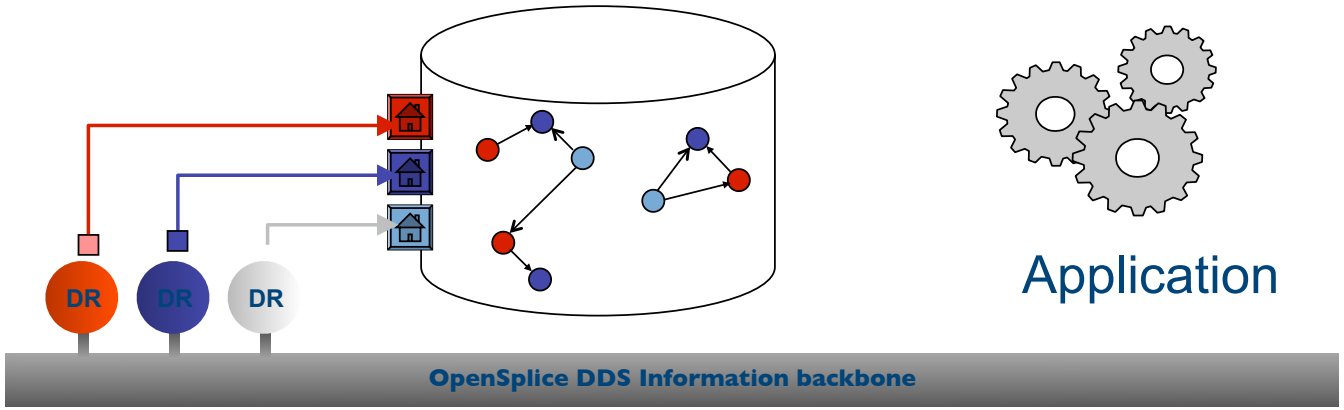
# Working with DLRL Objects




## Processing Updates

- DCPS updates arrive as separate samples at separate times.
- DLRL Updates are processed in 'update rounds':
  - ObjectHomes read all available samples from the DDS information backbone and update their corresponding objects in the Cache accordingly.
  - Objects are allocated once and their state is 'overwritten' on subsequent updates.
  - Therefore an Object always contains the latest available state.
  - Push mode: update rounds start when new data arrives. The application gets notified by Listeners.
  - Pull mode: the application can determine the start of each update round manually.

# Working with DLRL Objects

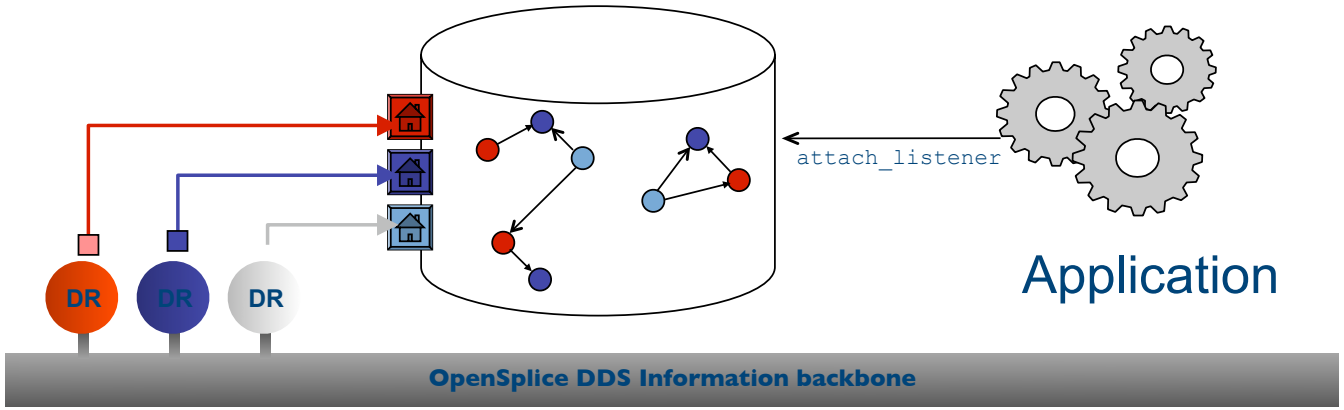



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

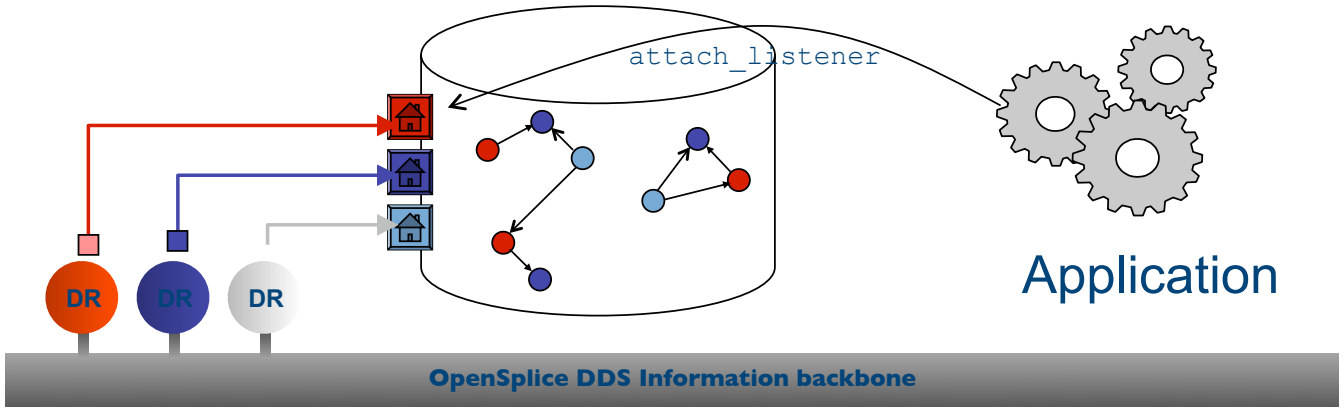



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

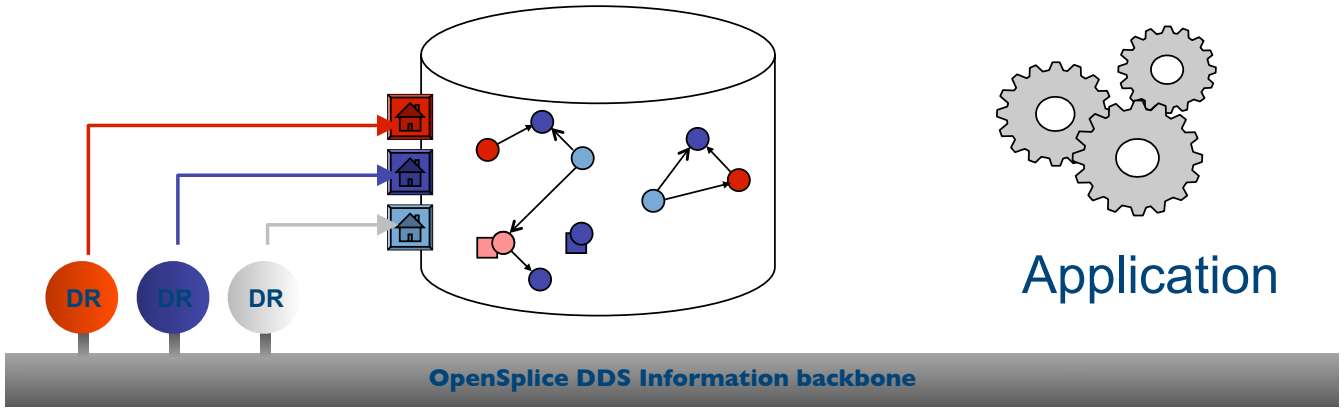



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

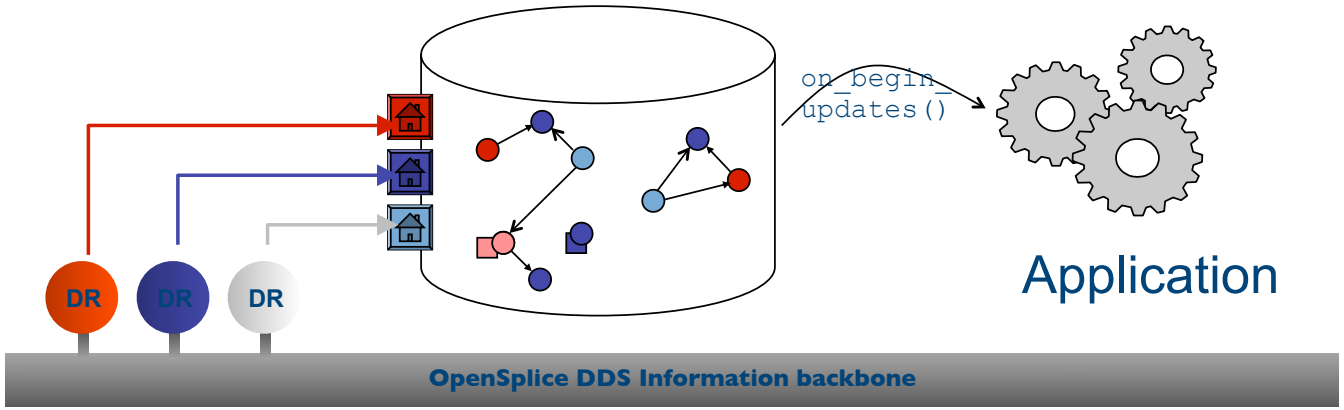



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

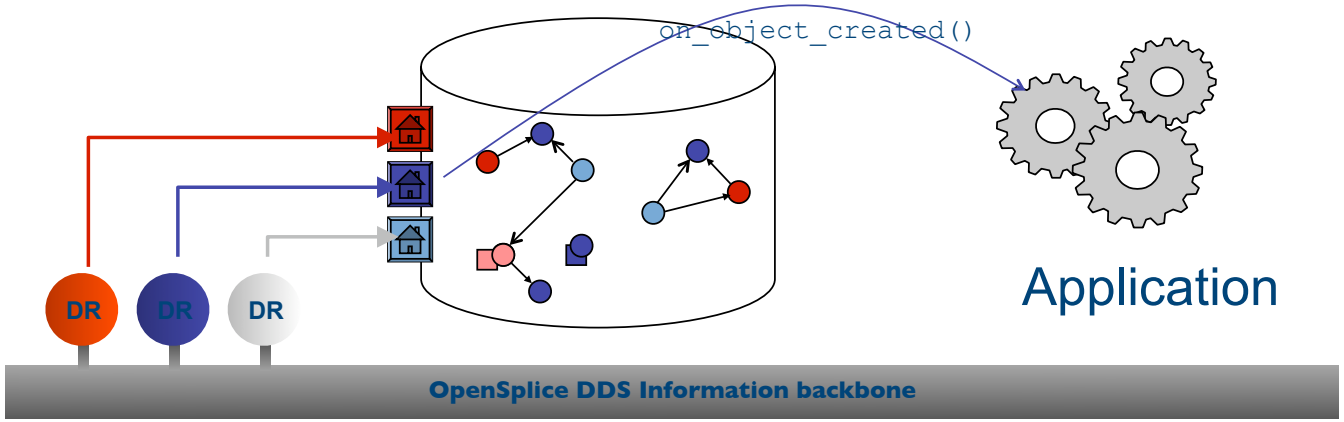



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

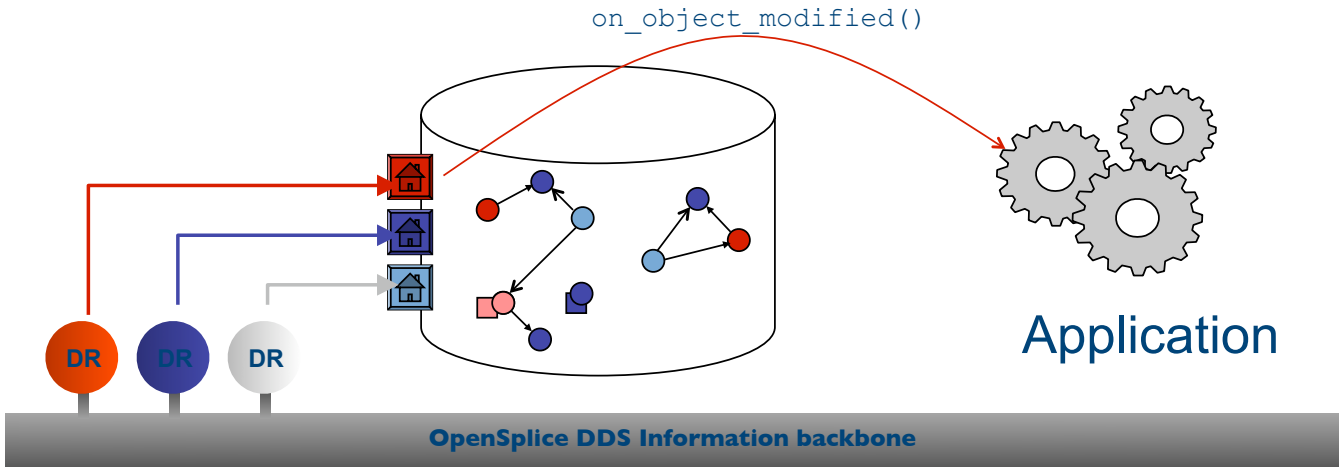



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

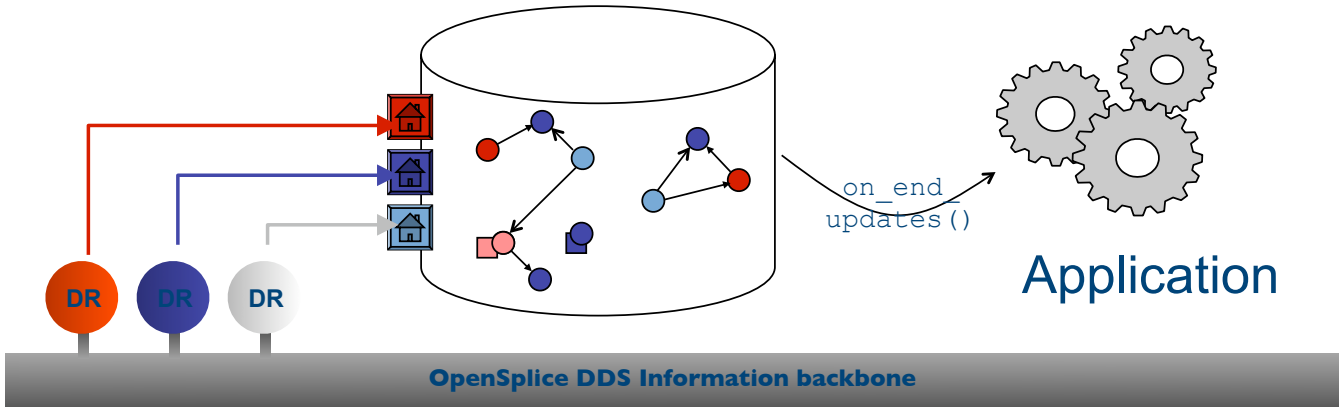



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

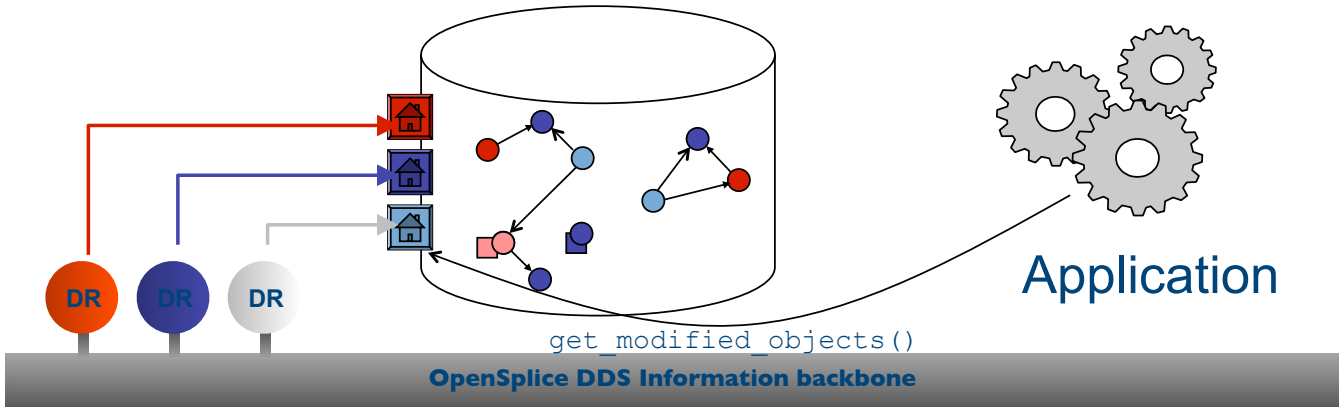



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

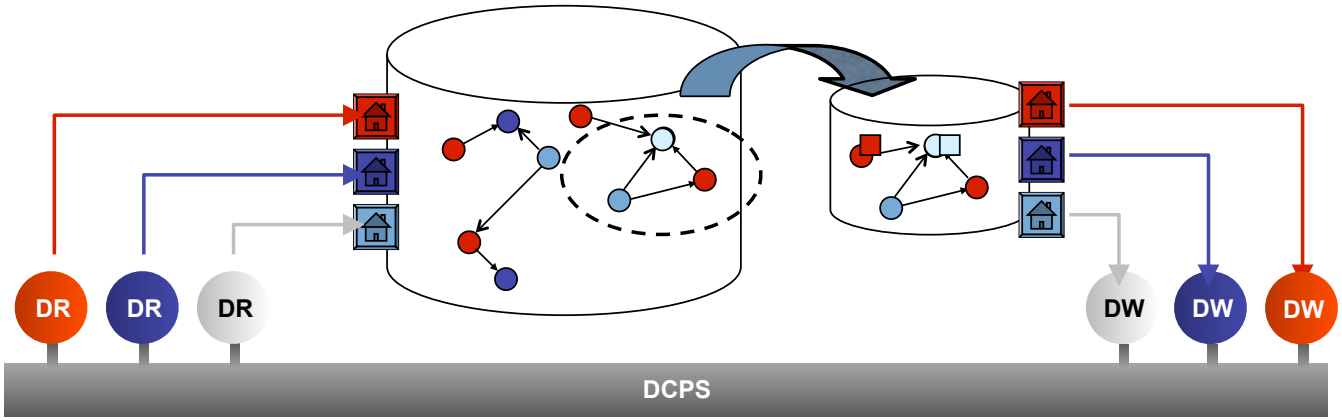



## Notifying the application

The Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object's state.
  - Listeners registered to the Cache indicate the start and end of each update round.
  - Listeners registered to the ObjectHome pass each modification back as a callback argument.
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects.
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round.

# Working with DLRL Objects

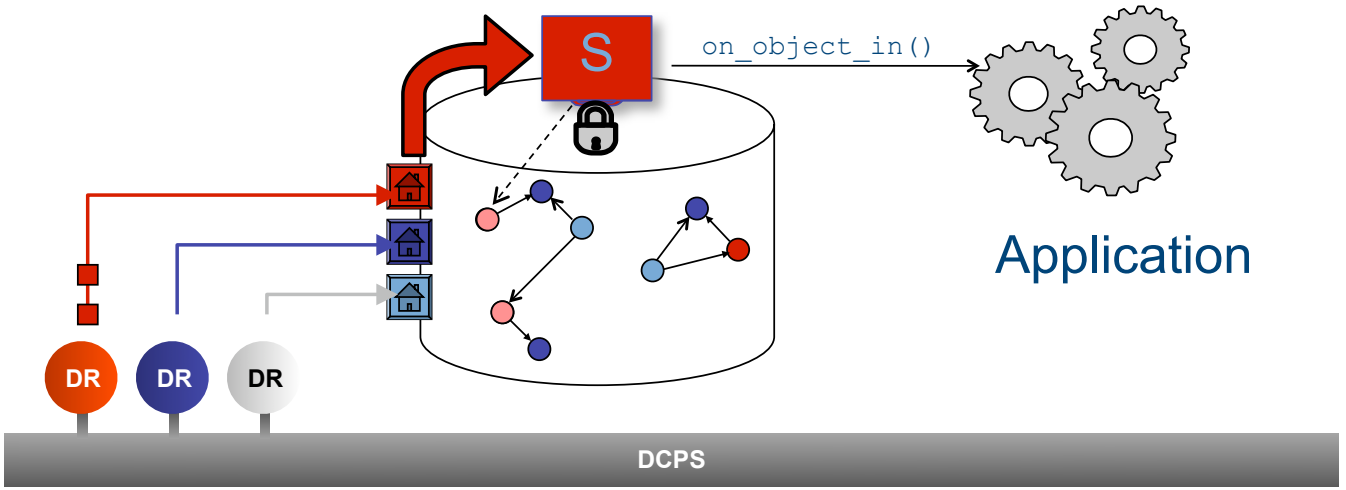



## Using snapshots

Some applications want to be able to modify or create certain objects:

- An initial set of Objects may be cloned into a writeable CacheAccess.
- Available objects may then be modified locally.
- New objects can be created in the CacheAccess as well.
- The 'write' operation instructs the ObjectHomes to write any modifications into the system.

# Working with DLRL Objects




## Creating and managing Selections

A Selection mechanism can keep track of subsets of information:

- Selections are created and managed by the ObjectHomes.
- A Criterion plugged into a Selection determines the boundaries of a subset:
  - A QueryCriterion determines boundaries based on an SQL statement.
  - A FilterCriterion determines boundaries based on user-defined callback filters.
- Selections can notify the application when objects enter and leave it.

# Putting it all Together

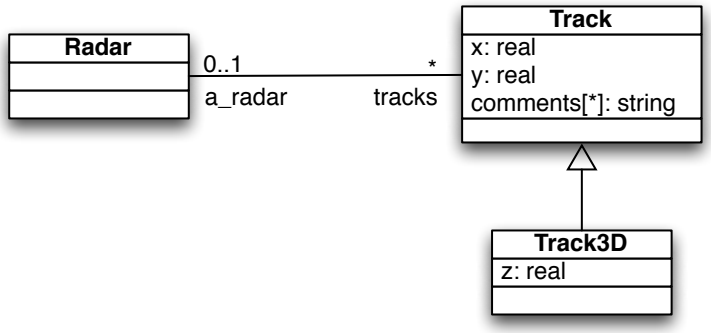
**Track.idl**

```
#include "dlrl.idl"
valuetype stringStrMap;      // StrMap<string>
valuetype TrackList;         // List<Track>
valuetype Radar;

valuetype Track : DLRL::ObjectRoot {
    public double       x;
    public double       y;
    public stringStrMap comments;
    public long         w;
    public Radar a_radar;
};

valuetype Track3D : Track {
    public double       z;
};

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




**mapping.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"/>
<classMapping name="Track">
    <local name="w"/>
</classMapping>
<associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
</associationDef>
</Dlrl>
```

# Putting it all Together

**main.cpp**

```
DDS::DomainParticipant_var dp;
DLRL::CacheFactory_var cf;
/*
 * Init phase
 */
DLRL::Cache_var c = cf->create_cache (WRITE_ONLY, dp);
RadarHome_var rh;
TrackHome_var th;
Track3DHome_var t3dh;
c->register_home (rh);
c->register_home (th);
c->register_home (t3dh);
c->register_all_for_pubsub();
// some QoS settings if needed
c->enable_all_for_pubsub();
```

**main.cpp (cont)**

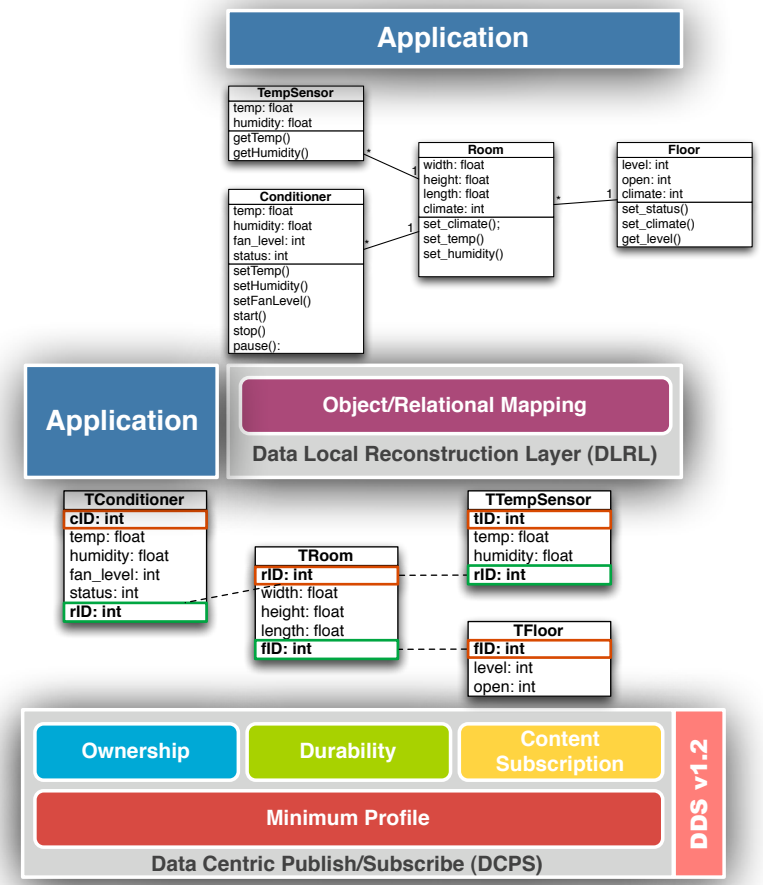
```
/*
 * Creation, modifications and publication
 */
Radar_var r1 = rh->create_object(c);
Track_var t1 = th->create-object (c);
Track3D_var t2 = t3dh->create-object (c);
t1->w(12); // setting of a pure local attribute
t1->x(1000.0); // some DLRL attributes settings
t1->y(2000.0);
t2->a_radar->put(r1);//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

# Agenda

- The Object/Relational Impedance Mismatch
- Why Hibernating DDS?
- DLRL: The Mysterious Acronym
- Hibernating DDS with DLRL
- **Concluding Remarks**

# Concluding Remarks

- DDS provides a very powerful facility for automating Object/Relational Mapping, namely the Data Local Reconstruction Layer (DLRL)
- The DLRL is an optional layer of the DDS v1.2 standard, and implementation exists, e.g., OpenSplice DDS, that support it for both C++ and Java
- The use of DLRL make it much more productive to write DDS application with Object Oriented Languages
- It is easy to get started with DLRL since it is quite similar to other popular Object/Relational Mapping Technologies such as Hibernate, JDO, etc.



# Online Resources

OpenSplice DDS

Delivering Performance, Openness, and Freedom

- http://www.opensplice.com/
- emailto:opensplicedds@prismtech.com

webex™

- http://www.opensplice.com

Blogger

- http://opensplice.blogspot.com

YouTube

- http://www.youtube.com/OpenSpliceTube

OMG OBJECT MANAGEMENT GROUP DDS

- http://www.dds-forum.org
- http://portals.omg.org/dds