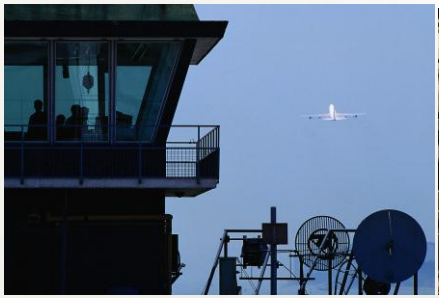
C++ now

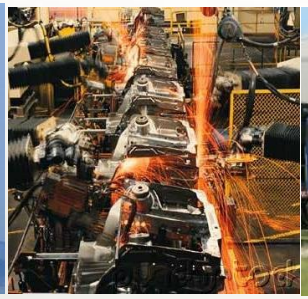
Aspen, CO
May 16, 2013

rti
Your systems. Working as one.

Standardizing the Data Distribution Service (DDS) API for Modern C++

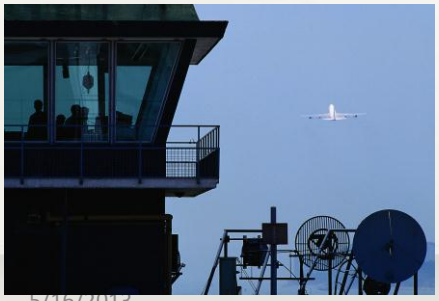
Sumant Tambe, Ph.D.
Senior Software Research Engineer
Real-Time Innovations, Inc.
www.rti.com

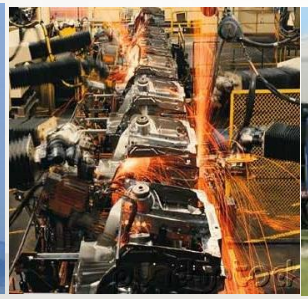




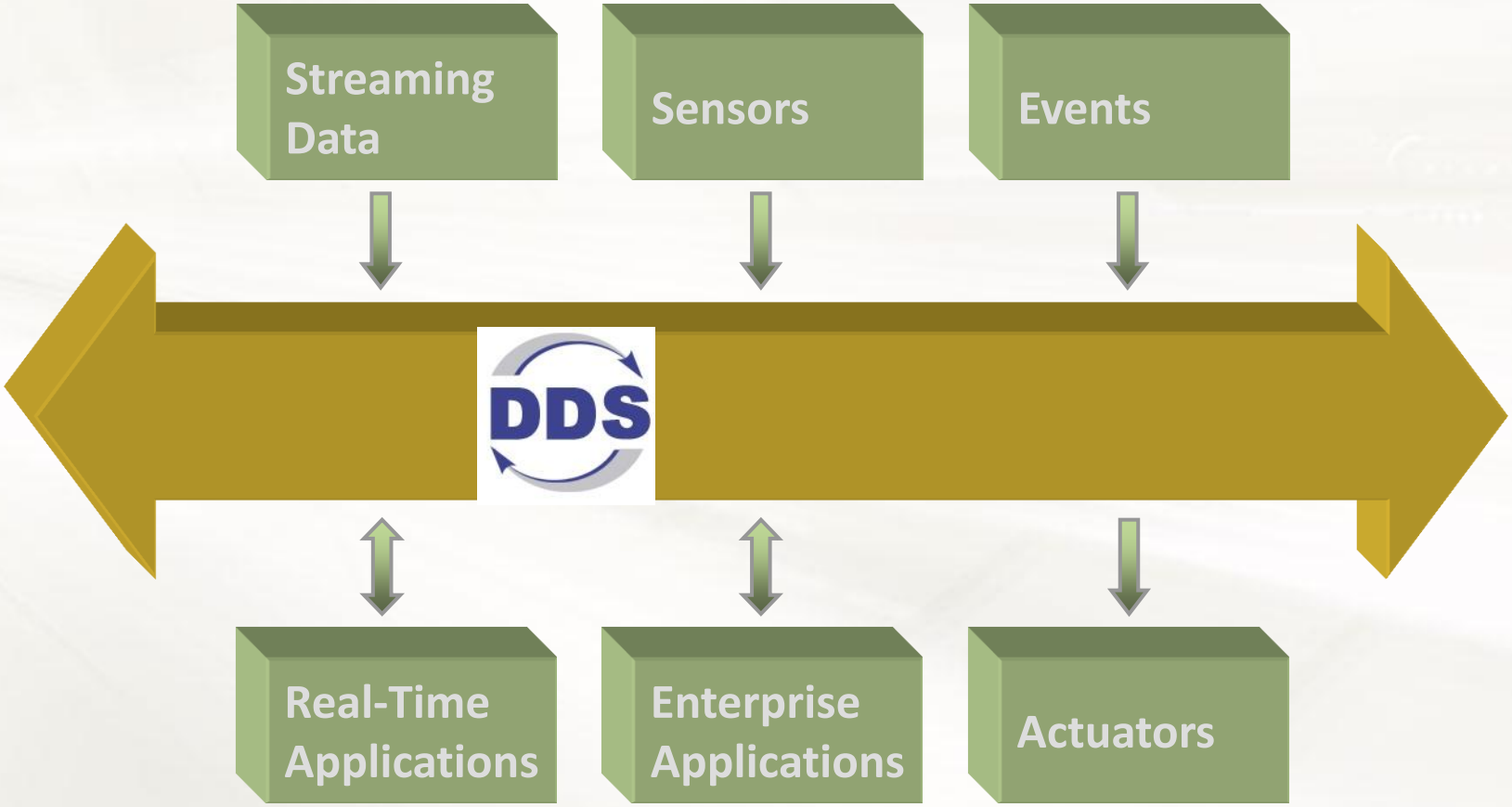
The Rise of Smart Systems

- DDS is for the systems that interact with the real world
  - Must adapt to changing environment
  - Cannot stop processing the information
  - Live within world-imposed timing



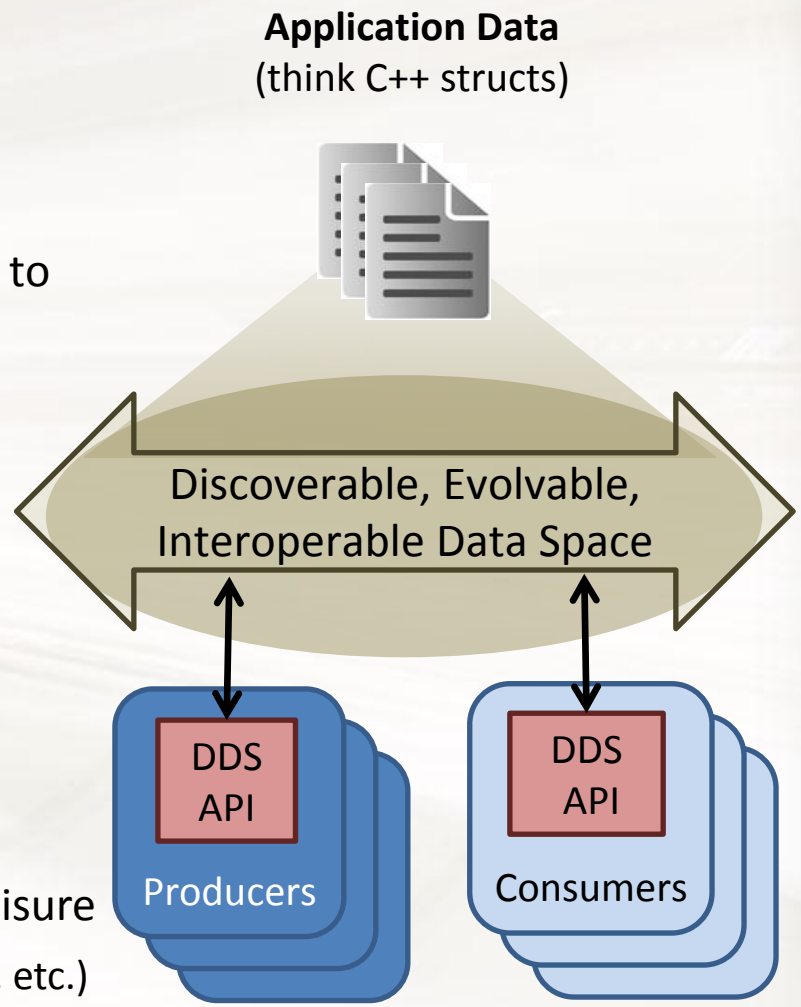


DDS: Standards-based Integration Infrastructure for Critical Applications

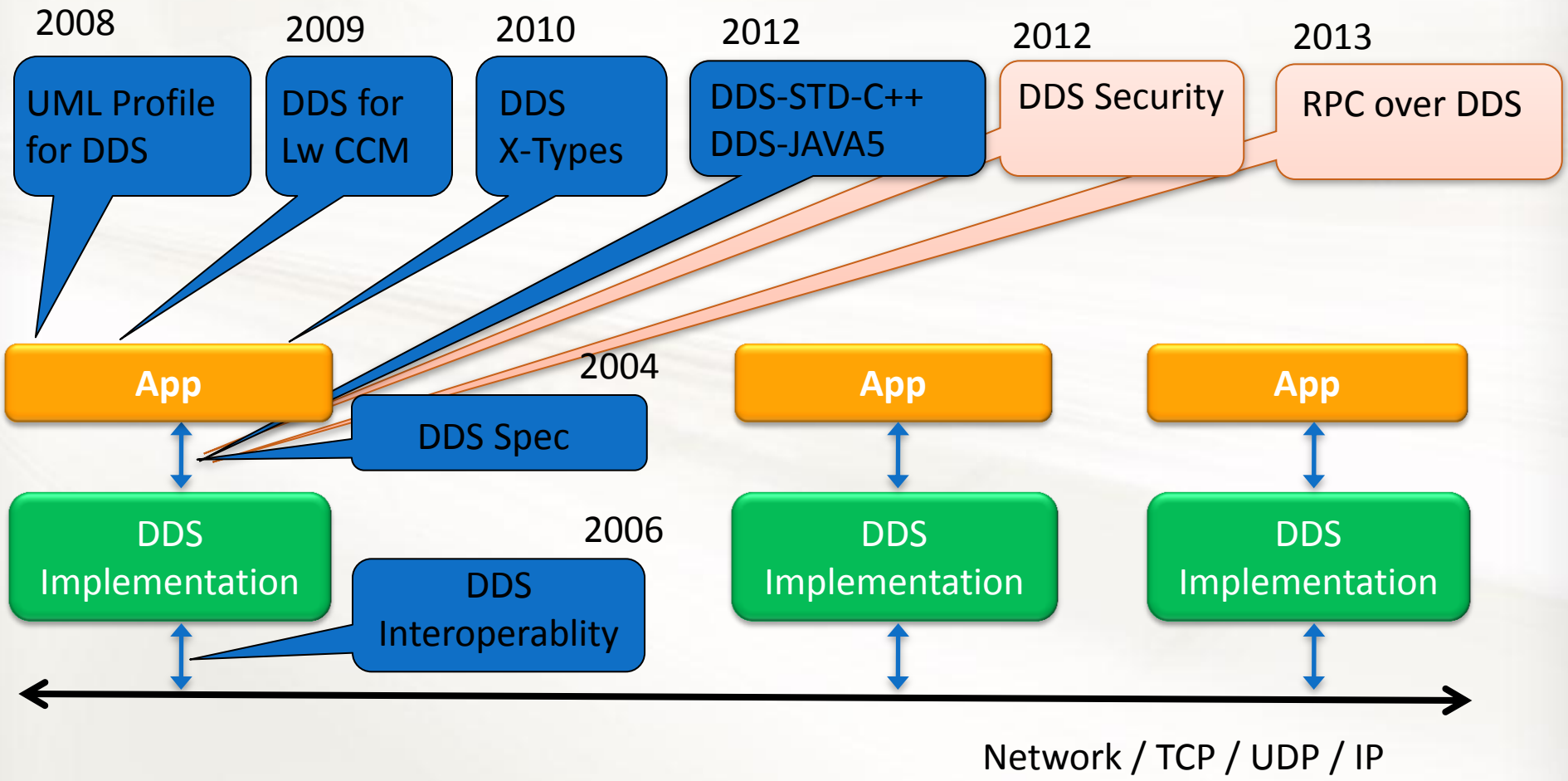


Pub/Sub Vs. Data Distribution

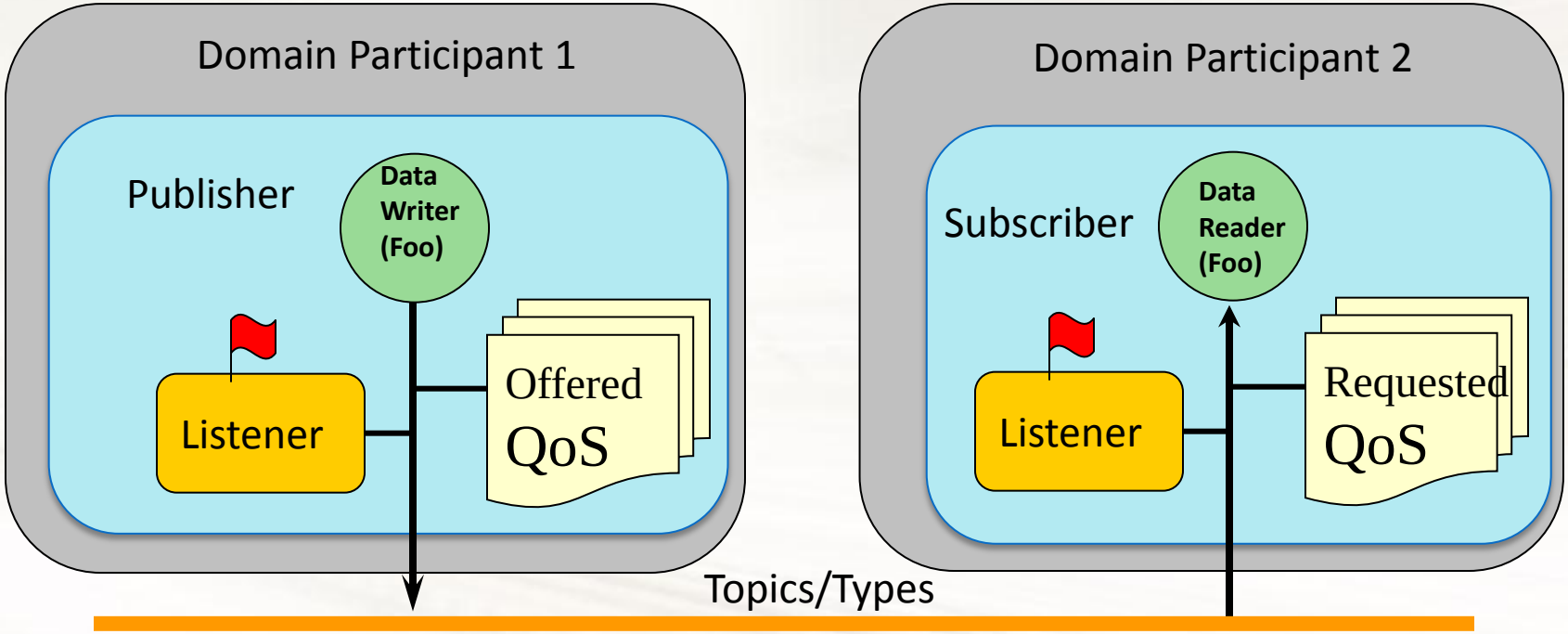
- Pub-Sub (message-centric)
  - Only messages. No concept of data object
  - Each message stands on its own
  - Messages must be delivered FIFO or according to some “priority” attribute
  - No (application visible) caching of data
  - Simple QoS: filters, durability, lifespan
- Data-Distribution (data-centric)
  - Full-fledged data model on wire
  - Messages represent update to data-objects
  - Data-Objects identified using a key
  - Middleware maintains state of data-objects
  - Objects are cached. Applications can read at leisure
  - Smart QoS (Reliability, Durability, History, Deadline, etc.)
  - Subsumes Pub-Sub



DDS
A Family of Standards

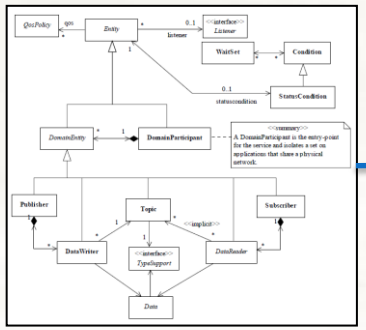


DDS Entities



- **Participants** scope the global data space (domain)
- **Topics** define the data-objects (collections of subjects)
- **DataWriters** publish data on Topics
- **DataReaders** subscribe to data on Topics
- **A Subscriber** may have many DataReaders
- **A Publisher** may have many DataWriters
- **QoS Policies** are used configure the system
- **Listeners** are used to notify the application of events

Classic DDS Language Bindings



UML Model of DDS
(Platform Independent)

DDS API in Interface Definition Language (IDL) (Platform-specific)

Standard
IDL Mapping

Java

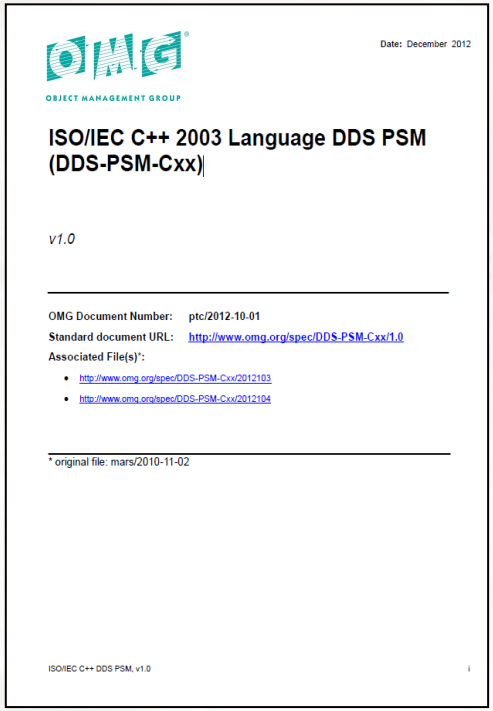
C/C++

C#

- IDL cannot capture programming language idioms
  - E.g., overloaded operators, static functions, new, templates, iterators, STL, meta-programming, exception-safety
- IDL-derived language mapping has limited C++ standard library integration
  - E.g., `char *` instead of `std:string`, `Sequence` instead of `std::vector`
- Awkward memory management for efficiency reasons
  - The "bucket" pattern: Provide the lower layer a place to put the result. Pollutes API
- Not 100% portable
  - Some DDS types left to implementers to decide
  - `int` and `struct` initialization syntax is different prior to C++11

C++ Language DDS PSM Motivations

- Intuitive API
  - Provide better integration with the C++ programming language and the standard library
  - An API that is efficient, expressive, easy-to-use, easy-to-learn, and exception-safe
  - Works well with “intellisense” editors
- Ensure 100% portability
  - Drop in replacement of vendor implementations; ideally just a recompile and relink
  - Standard OMG-managed header files can be used to test compliance and portability
- Extensible
  - Implementers can extend the API
    - New quality-of-service (QoS) policies, new members in the standard policies, etc.
  - The extensions are syntactically distinct
- Forward-looking
  - Make special provisions for C++11


OBJECT MANAGEMENT GROUP

Date: December 2012

ISO/IEC C++ 2003 Language DDS PSM (DDS-PSM-Cxx)

v1.0

OMG Document Number: ptc/2012-10-01

Standard document URL: http://www.omg.org/spec/DDS-PSM-Cxx/1.0

Associated File(s)*:

- http://www.omg.org/spec/DDS-PSM-Cxx/2012103
- http://www.omg.org/spec/DDS-PSM-Cxx/2012104

* original file: mars/2010-11-02

ISO/IEC C++ DDS PSM, v1.0

OMG Finalization Task Force Makeup

Gallium

Track DataWriter Example

```
class Track; // some user-defined type (often generated)

dds::domain::DomainParticipant dp(0);
dds::topic::Topic<Track> topic(dp, “track-topic”);
dds::pub::Publisher pub(dp);
dds::pub::qos::DataWriterQos dwqos =
  pub.default_writer_qos() << Reliability::Reliable()
                           << History::KeepLast(10)
                           << Durability::TransientLocal();
dds::pub::DataWriter<Track> dw (pub, topic, dwqos);

Track t = { 0xDEAD, “tank” }; // track id and vehicle-type
for(;;) {
  dw.write(t);
}
```

Track DataReader Example

```
try {
  dds::domain::DomainParticipant dp(0);
  dds::topic::Topic<Track> topic(dp, “track-topic”);
  dds::sub::Subscriber sub(dp);

  dds::sub::qos::DataReaderQos drqos =
    sub.default_reader_qos() << Reliability::BestEffort()
                             << History::KeepLast(5);
                             << Durability::TransientLocal();

  dds::sub::DataReader<Track> dr (sub, topic, drqos);

  LoanedSamples<Track> data = dr.read();
  std::for_each(data.begin(), data.end(), printTracks);
}
catch (const dds::core::PreconditionNotMetError&) { ... }
catch (const dds::core::Exception& ex) { ... }
catch (const std::exception& ex) { ... }
catch (...) { ... }
```

Mapping User-Defined Data Types in C++03

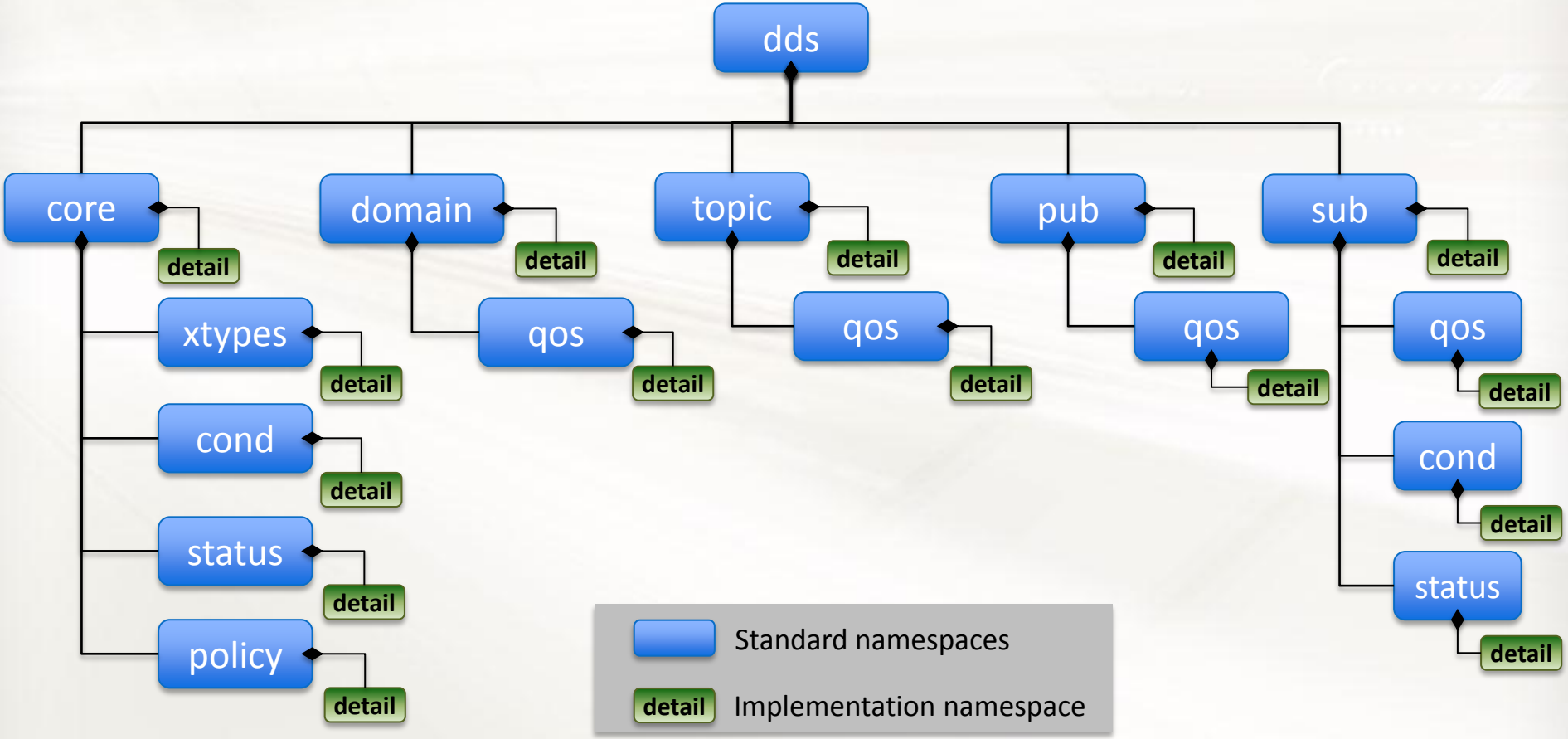
User-Defined IDL	C++03 Representation (almost always generated)

```
typedef sequence<octet>
    plot_t;

struct Track {
  string id;
  long lat;
  long lon;
  long alt; //@optional
  plot_t plot;
};
```

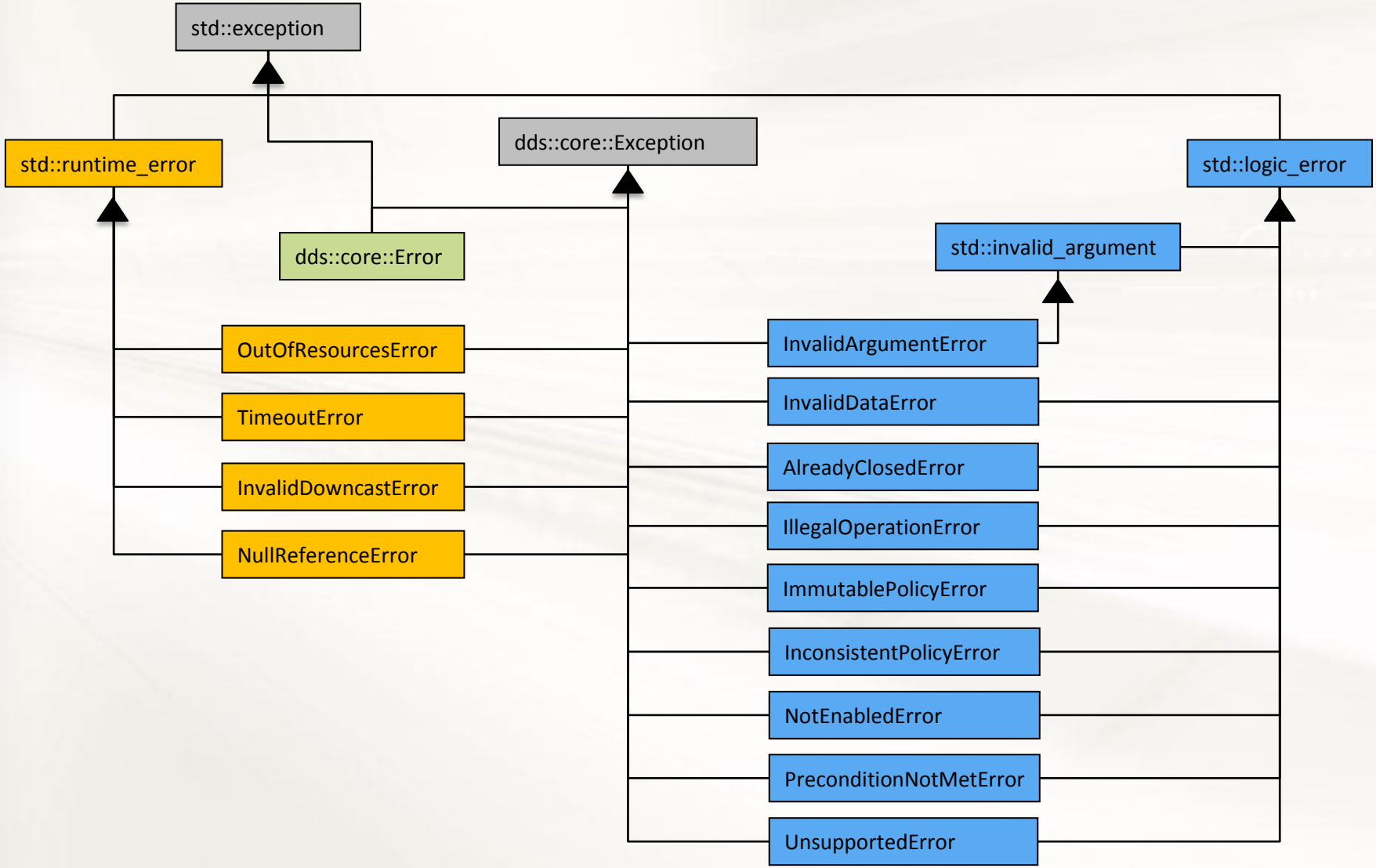
```
typedef std::vector<uint8_t> plot_t;

class Track {
private:
  // state representation is implementation dependent
public:
  typedef smart_ptr_traits<plot_t>::ref_type plot_ref_t;

  Track();
  Track(const std::string & id,
        int32_t lat, int32_t lon, int32_t alt,
        std::vector<uint8_t> * plot);

  std::string & id();
  const std::string & id() const;
  void id(const std::string &);

  int32_t lat() const;
  void lat(int32_t);

  int32_t lon() const;
  void lon(int32_t);

  dds::core::optional<int32_t> alt() const;
  void alt(int32_t);
  void alt(const dds::core::optional<int32_t> &);

  plot_ref_t & plot();
  const plot_ref_r & plot() const;
  void plot(const plot_ref_t &);
};
```

ISO/IEC C++ 2003 Language DDS PSM

- Organization
  - C++ namespaces group relevant classes
  - Fine grain `#includes` also possible



Exceptions for Error Handling



Instantiating the Standard API

- The standard uses DELEGATEs—provided by vendors–to instantiate the concrete types
  - Compliant implementations must not change the standard API
  - Vendor-specific extensions are possible but accessible only through the DELEGATEs
  - The standard provides a well-defined syntax to access the extensions
  - Vendor-specific types appear only in the `detail` namespace

```
namespace rti {
  class InstanceHandleImpl; // vendor-specific
}

namespace dds {  namespace core {
  template <typename DELEGATE> class TInstanceHandle { ... }; // standard
} }

namespace dds { namespace core { namespace { detail
  typedef dds::core::TInstanceHandle<rti::InstanceHandleImpl> InstanceHandle; // vendor-specific
} } }

namespace dds { namespace core {
  typedef dds::core::detail::InstanceHandle InstanceHandle; // bring name in the standard namespace.
} }
```

Accessing Vendor-specific Extensions

- Vendor-specific extensions are possible but accessible only through the DELEGATEs
- The standard provides a well-defined syntax to access the extensions
  - Dot: `obj.method()` for standard API
  - Arrow: `obj->extension_method()` for extensions

```
namespace rti {
  class InstanceHandleImpl {
    bool is_nil() const; // standard
    void rti_extension(); // extension
  }; // vendor-specific
}

namespace dds {  namespace core {
  template <typename DELEGATE>
  class TInstanceHandle // standard
  {
    DELEGATE * operator -> ();
    const DELEGATE * operator -> () const;
    bool is_nil() const;
  }; // standard
} }

int main(void) {
  dds:core::InstanceHandle handle = ...;
  handle.is_nil(); // standard
  handle->rti_extension(); // extension
}
```

Instantiating the Standard API (templates)

- Some DELEGATEs are themselves templates
- The type parameter is provided by the user
  - The standard must forward the vendor-specific templates to the standard namespaces

```
namespace rti {
  template <class T>
  class DataReader { ... }; // vendor-specific
}

namespace dds {  namespace sub { namespace detail {
  using rti::DataReader;
} } }
```

C++11 template aliases would allow different names

```
template<class T>
using DataReader = rti::DataReaderImpl<T>;
```

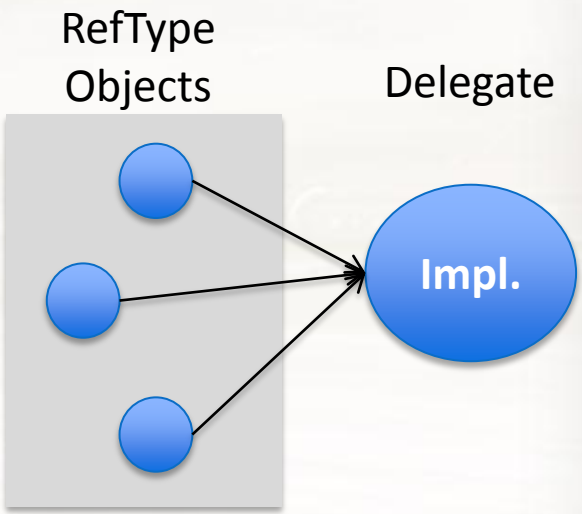
Can't change!

```
namespace dds { namespace sub
  template <typename T,
            template <typename Q> class DELEGATE = dds::sub::detail::DataReader>
  class DataReader { ... };  // standard
} }
```

```
int main(void) {
  dds::sub::DataReader<Foo> dr; // user-code
}
```

DDS C++ PSM Object Model

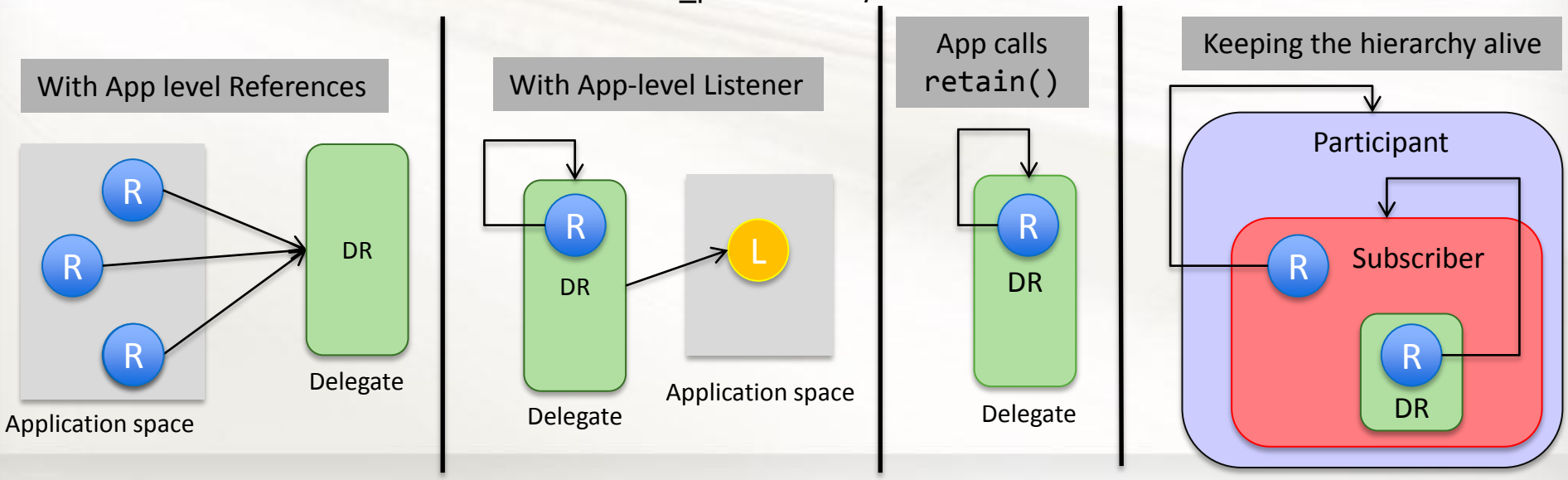
- Reference Types
  - E.g., `DomainParticipant, Subscriber, Publisher, DataReader, DataWriter`, etc.
  - Shallow copy semantics; Identical if the pointers match
  - Inherit from `core::Reference<DELEGATE>`
    - Manage memory for DDS entities automatically
    - Reference counted—similar to `boost::shared_ptr`
    - Works with `core::WeakReferenceReference<DELEGATE>`
      - Similar to `boost::weak_ptr`
      - Can be constructed from strong references only
      - You can either check if the weak reference is valid or get the strong reference from it



- Value Types
  - E.g., Policies (History, Reliabiliy, Durability, etc.), QoS objects, InstanceHandle
  - Deep copy semantics; Identical if the contents match
  - Inherit from `core::Value<DELEGATE>`

Managing The Entity Lifecycle

- Entity Survival rules!
  - Any entity to which the application has a direct reference (but not a WeakReference) is still in use.
  - Any entity with a non-null listener is still in use.
  - Any entity that has been explicitly `retain`ed is still in use. Application can retrive the entity anytime using `dds::pub::find, dds::sub::find`, etc.
  - If a child object is in use, the parent is also in use. (E.g., All `DataReaders` keep the corresponding `Subscribers` and the `DomainParticipant` alive)
  - Regardless of references/listeners/`retain`, `close` terminates the entity.
- DDS PSM C++ relies on the shared pointer idiom for entity lifecycle
  - `dds::core::Reference` uses `shared_ptr` internally



DDS Real-Time QoS Policies

	QoS Policy	QoS Policy	
Volatility	DURABILITY	USER DATA	User QoS
	HISTORY	TOPIC DATA	
	READER DATA LIFECYCLE	GROUP DATA	
	WRITER DATA LIFECYCLE	PARTITION	Presentation
Infrastructure	LIFESPAN	PRESENTATION	
	ENTITY FACTORY	DESTINATION ORDER	
	RESOURCE LIMITS	OWNERSHIP	Redundancy
	RELIABILITY	OWNERSHIP STRENGTH	
Delivery	TIME BASED FILTER	LIVELINESS	
	DEADLINE	LATENCY BUDGET	Transport
	CONTENT FILTERS	TRANSPORT PRIORITY	

Supporting QoS Extensibility

- Vendors provide many additional QoS policies or additional attributes for the standard QoS policies
  - For example, RTI Connext™ DDS provides extension attributes to standard QoS policies
    - Reliability::max_blocking_time
    - Reliability::acknowledgement_kind
    - History::refilter_qos
  - Also, RTI Connext™ DDS provides many extension QoS
    - Control receiver thread pool
    - Batching of data samples
    - Various transports, and more…
- The standard wants to provide a **consistent, extensible** API that accommodates the standard as well as extension QoS policies.

Extensible Qos via EntityQos

- An EntityQos object represents a heterogenous "collection" of policies
  - typedef EntityQos<rti::DataReaderQos> DataReaderQos
  - typedef EntityQos<rti::DataWriterQos> DataWriterQos
- Provides a DSL using the overloaded operators
- For example,

```
datawriter_qos << History::KeepAll();

dds::core::policy::History h;
dds::core::policy::Deadline d;
datawriter_qos >> h >> d;
```

- Extensible naturally to vendor-specific Qos policies

```
template <typename DELEGATE>
class dds::core::EntityQos
      : public dds::core::Value<DELEGATE>
{
public:

  template <typename POLICY>
  EntityQos& policy(const POLICY& p);

  template <typename POLICY>
  EntityQos& operator << (const POLICY& p);

  template <typename POLICY>
  const POLICY& policy() const;

  template <typename POLICY>
  POLICY& policy();

  template <typename POLICY>
  const EntityQos& operator >> (POLICY& p) const;
};
```

Is EntityQos too Generic?

Qos Policy	Applicability
DURABILITY	**T, DR, DW**
DURABILITY_SERVICE	**T, DW**
LIFESPAN	**T, DW**
HISTORY	**T, DR, DW**
PRESENTATION	**P, S**
RELIABILITY	**T, DR, DW**
PARTITION	**P, S**
DESTINATION_ORDER	**T, DR, DW**
OWNERSHIP	**T, DR, DW**
OWNERSHIP_STRENGTH	**DW**
DEADLINE	**T, DR, DW**
LATENCY_BUDGET	**T, DR, DW**
TRANSPORT_PRIORITY	**T, DW**
TIME_BASED_FILTER	**DR**
RESOUCE_LIMITS	**T, DR, DW**
USER_DATA	**DP, DR, DW**
TOPIC_DATA	**T**
GROUP_DATA	**P, S**

T=Topic, **DR**=DataReader, **DW**=DataWriter,
P=Publisher, **S**=Subscriber, **DP**=DomainParticipant

- An `EntityQos` object represents a collection of policies
  - `typedef EntityQos<rti::DataReaderQos> DataReaderQos`
  - `typedef EntityQos<rti::DataWriterQos> DataWriterQos`
- Not all policies can be combined with all Qos objects (see table)
- How to prevent setting policies that do not make sense for a qos?
  - For example, prevent assignment of partition to DWQos at compile-time
  - ~~`datawriter_qos << Partition("A-stocks");`~~

Constraining EntityQos

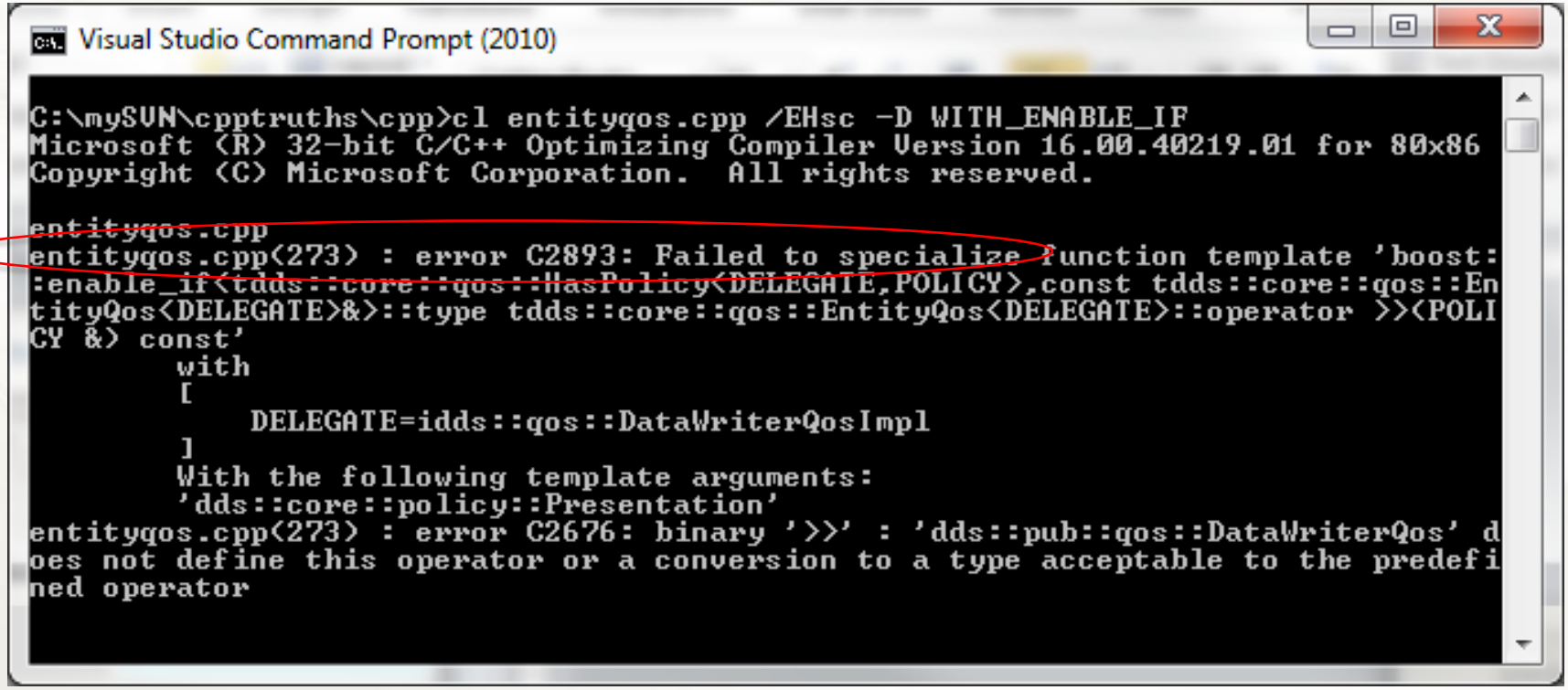
- Define "marker" interfaces `ForDataReader`, `ForDataWriter`, ForTopic etc.
- Have policies inherit from appropriate marker interfaces
  - `class Reliability : public ForDataReader, ForDataWriter, ForTopic { ... }`
  - `class Partition : public ForPublisher, ForSubscriber { ... }`
- Use SFINAE or static_assert to disallow invalid combinations

```
template <typename DELEGATE>
class dds::core::EntityQos
: public dds::core::Value<DELEGATE>
{
public:
  template <typename POLICY>
  EntityQos&
  operator << (const POLICY& p);

  // ...
};
```

```
template <typename DELEGATE>
class dds::core::DataReaderQos
: public dds::core::EntityQos<DELEGATE>
{
public:
  template <typename POLICY>
  typename enable_if<is_base_of<ForDataReader, POLICY>,
                     DataReaderQos &>::type
  operator << (const POLICY& p) { ... }

  // alternatively

  template <typename POLICY>
  DataReaderQos &
  operator << (const POLICY& p)
  {
    OMG_STATIC_ASSERT(is_base_of<ForDataReader, POLICY>::Value);
  }
};
```

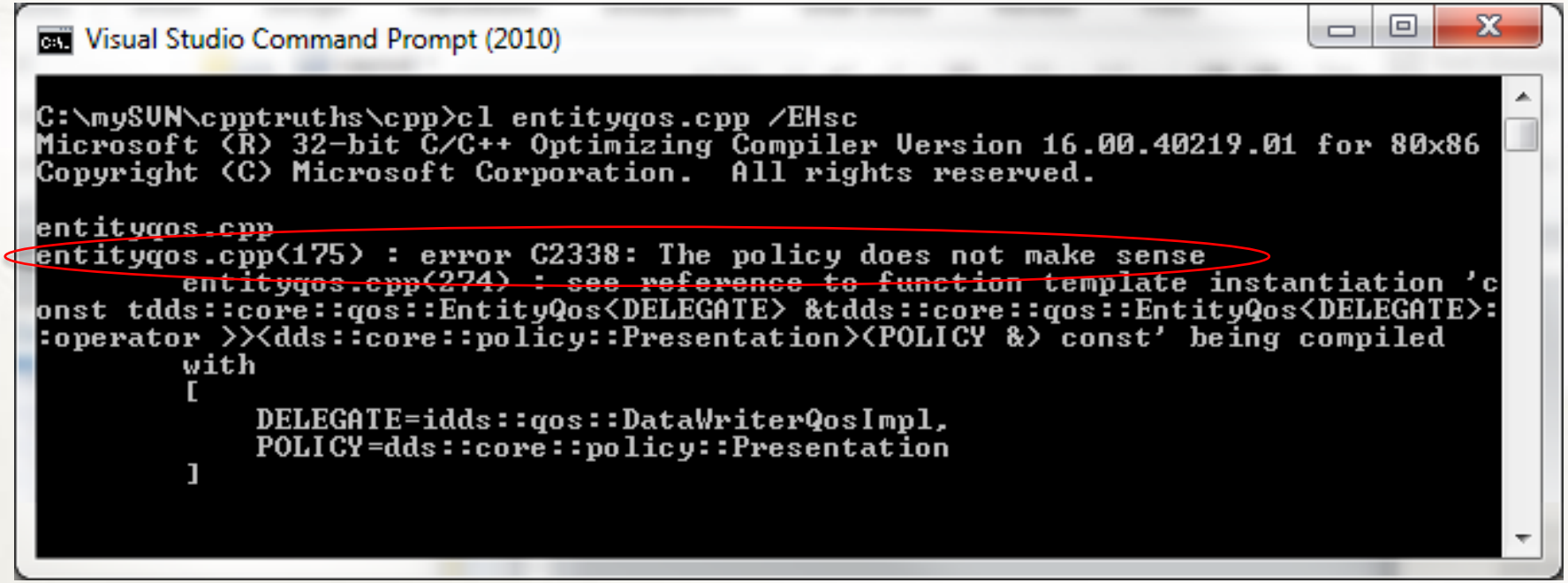
Choice of Error: SFINAE

- SFINAE (Substitution Failure Is Not An Error)
  - Remove the function out of the overload set
  - Pros: The error is shown in the user code as opposed to the library code
  - Con: Error: “Failed to specialize” – not very informative



Choice of Error: static_assert

- Insert a `static_assert` with a descriptive message.
  - Pros: Clear message. However, can't be customized. It would be nice to do printf-style static_assert.
  - Con: The error is shown in the library implementation
  - Con: C++11 only



Reading/Taking Data

- The API provides "data-centric" access
  - DDS provides built-in caching of data for future use
  - **`read`**—means access data but keep it in the middleware buffer
    - Internal queue does not grow unbounded because QoS policies limit resource usage
  - **`take`**—means access data and remove it from the middleware buffer
  - Query/Filter data based on
    - Sample state (read or not read)
      - *"Have I read this sample before or not?"*
    - View state (viewed or not viewed)
      - *"Is this a new tank or I've seen it before?"* – based on *key*
    - Instance state (alive, not alive_disposed, not_alive_no_writers)
      - *"Is there anyone out there talking about that specific tank?"*
    - A specific instance (instance_handle)
      - *"What's the latest update (location) on this specific tank?"*
    - Next instance (analogous to an iterator of instances)
      - "Get me data on each tank in the convoy." – based on some total ordering of the *keys*
    - Samples that match an expression (query condition)
      - *"Tell me about the tanks within 10 miles of (x,y): "lat = x && lon = y && radius == 10".*

Reading/Taking Data

- 6 basic functions

```
template <typename T, template <typename Q> class DELEGATE>
class dds::sub::DataReader
{
  LoanedSamples<T> read(); // LoanedSamples<T> provides begin/end
  LoanedSamples<T> take();

  template <typename SamplesFWIterator>
  uint32_t read(SamplesFWIterator sfit, uint32_t max_samples); // forward iterator

  template <typename SamplesFWIterator>
  uint32_t take(SamplesFWIterator sfit, uint32_t max_samples); // forward iterator

  template <typename SamplesBIIterator>
  uint32_t read(SamplesBIIterator sbit); // back-insert iterator

  template <typename SamplesBIIterator>
  uint32_t read(SamplesBIIterator sbit); // back-insert iterator
};
```

Reading/Taking/Querying Data

- Fluent interface in C++
  - Method chaining

```
std::vector<Track> tracks = ...
DataReader<Track> dr = ...
InstanceHandle handle = dr.lookup_instance(Track(id));

dr.select()
      .instance(handle)
      .content(Query(dr, "x < 25 AND y > 10"))
      .state(DataState::new_data())
      .max_samples(100)
  .take(std::back_inserter(tracks));
```

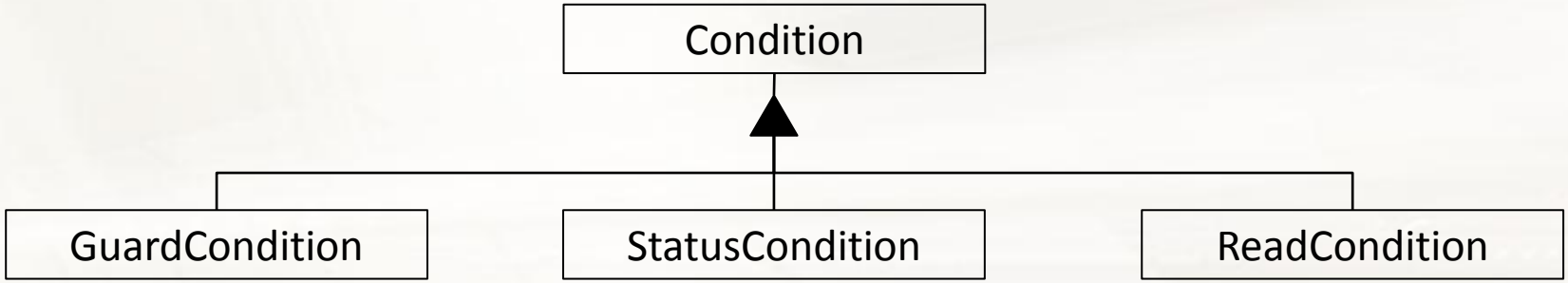
All 6 basic forms are applicable

Abstracting Queries using Selector

- The Selector separates **how** to read from **what** to read
  - N+M instead of N*M
  - **How:** LoanedSamples, Forward iterators, back-insert iterators
  - **What:** specified using the query object
  - Uses the method-chaining idiom

```
template <typename T, template <typename Q> class DELEGATE>
class dds::sub::DataReader
{
  class Selector {
  public:
    Selector(DataReader& dr);

    Selector& instance(const dds::core::InstanceHandle& h);
    Selector& next_instance(const dds::core::InstanceHandle& h);
    Selector& state(const dds::sub::status::DataState& s);
    Selector& content(const dds::sub::Query& query);
    Selector& max_samples(uint32_t n);

    // Selector also has the basic read/take functions
    // which delegate to the DataReader.
  };
};
```

Reading Data using Conditions and WaitSets



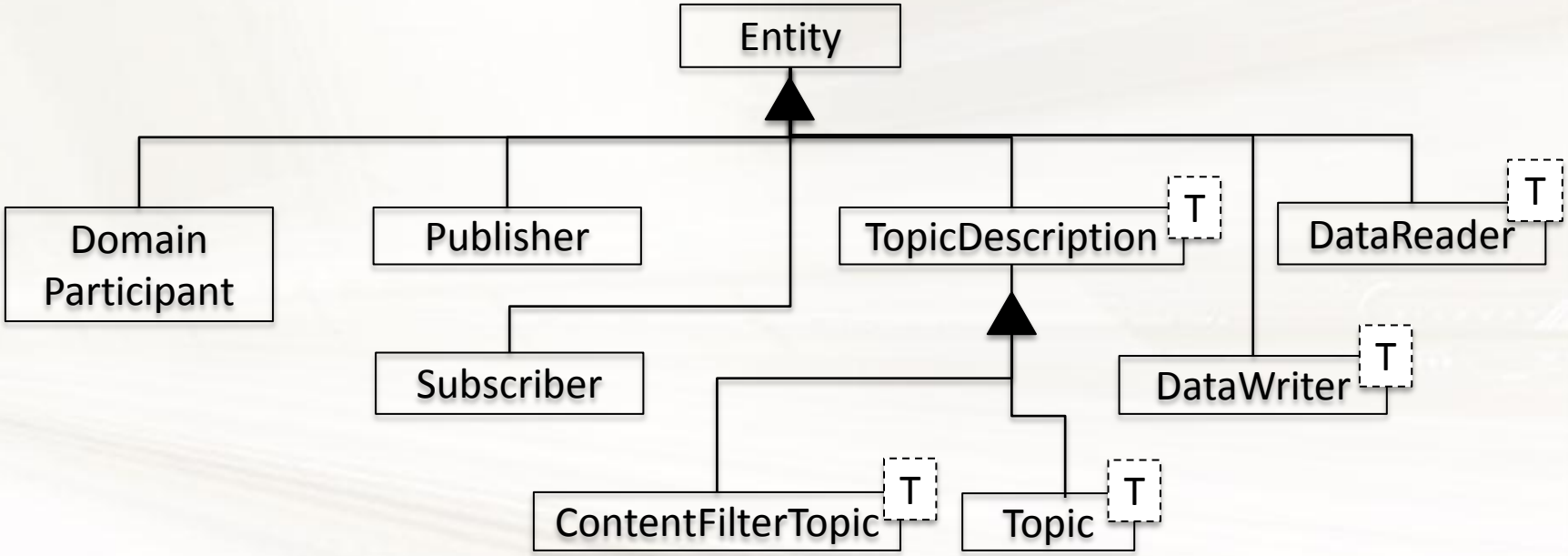
Provides a way for an application to signal conditions to other threads

Wakes up the application based on status m/w entities. E.g.,
DATA_AVAILABLE
LIVELINESS_LOST
SAMPLE_LOST
DEADLINE_MISSED

Wakes up the application based on the status of the sample data. E.g.,
NOT_VIEWED_SAMPLE
NEW_VIEW_STATE
ALIVE_INSTANCE_STATE

- WaitSets and conditions work together
  - Multiple conditions can be attached to a WaitSet
  - WaitSet unblocks when one or more conditions become active

Managing DDS Entities Polymorphically

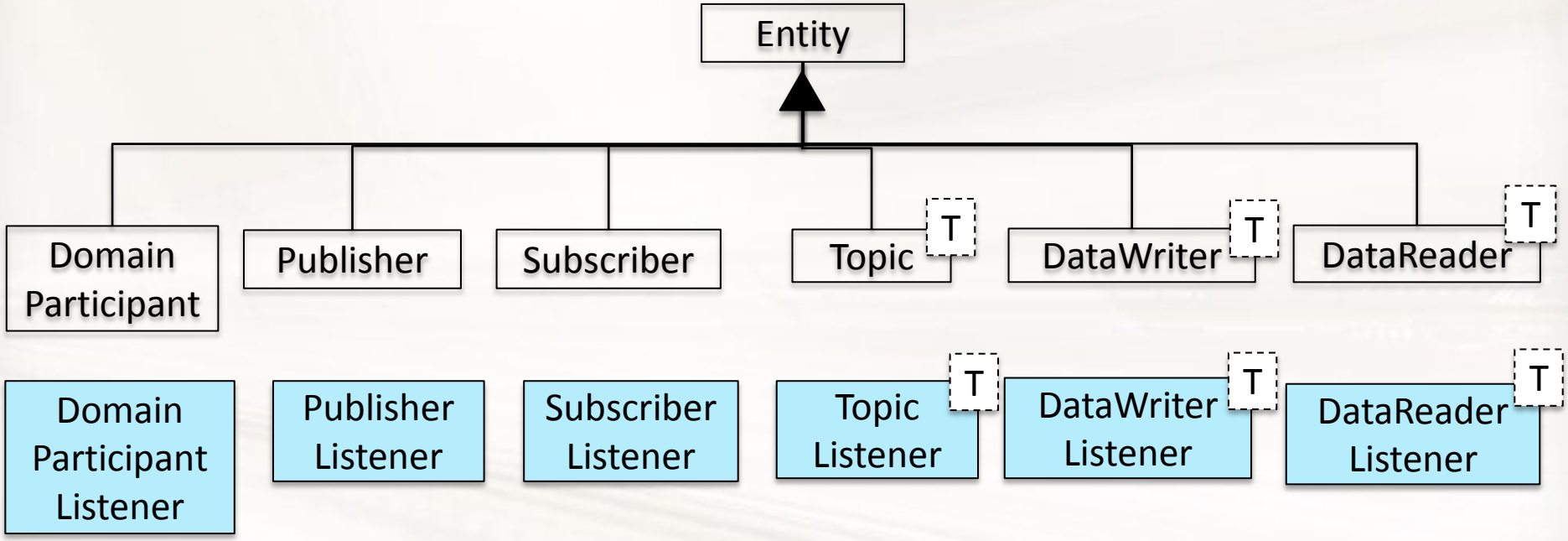



```
template <typename DELEGATE>
class dds::core::TEntity : public dds::core::Reference<DELEGATE> {
public:
  void enable();
  const dds::core::status::StatusMask status_changes();
  const dds::core::InstanceHandle instance_handle() const;
  void close();
  void retain();
};
```

Not virtual

- Polymorphism is an implementation detail!
  - Managed internally by the DELEGATEs using an analogous hierarchy of `EntityImpl` objects

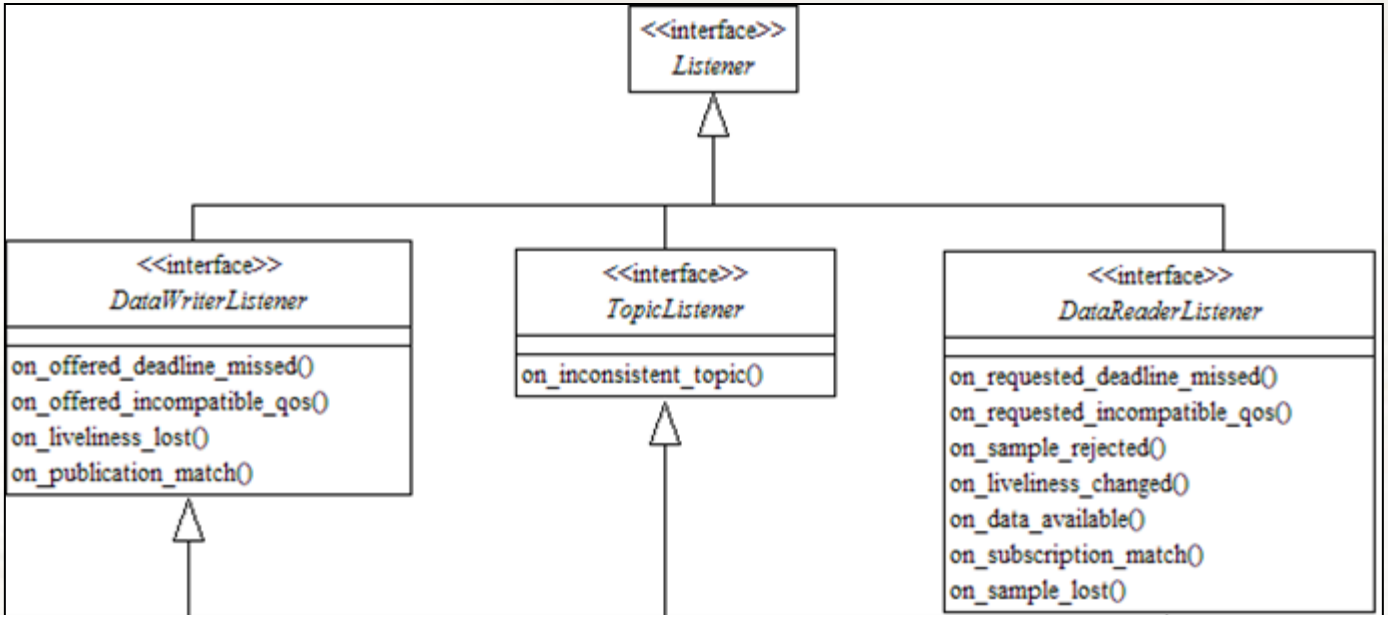
DDS Entities and Listeners



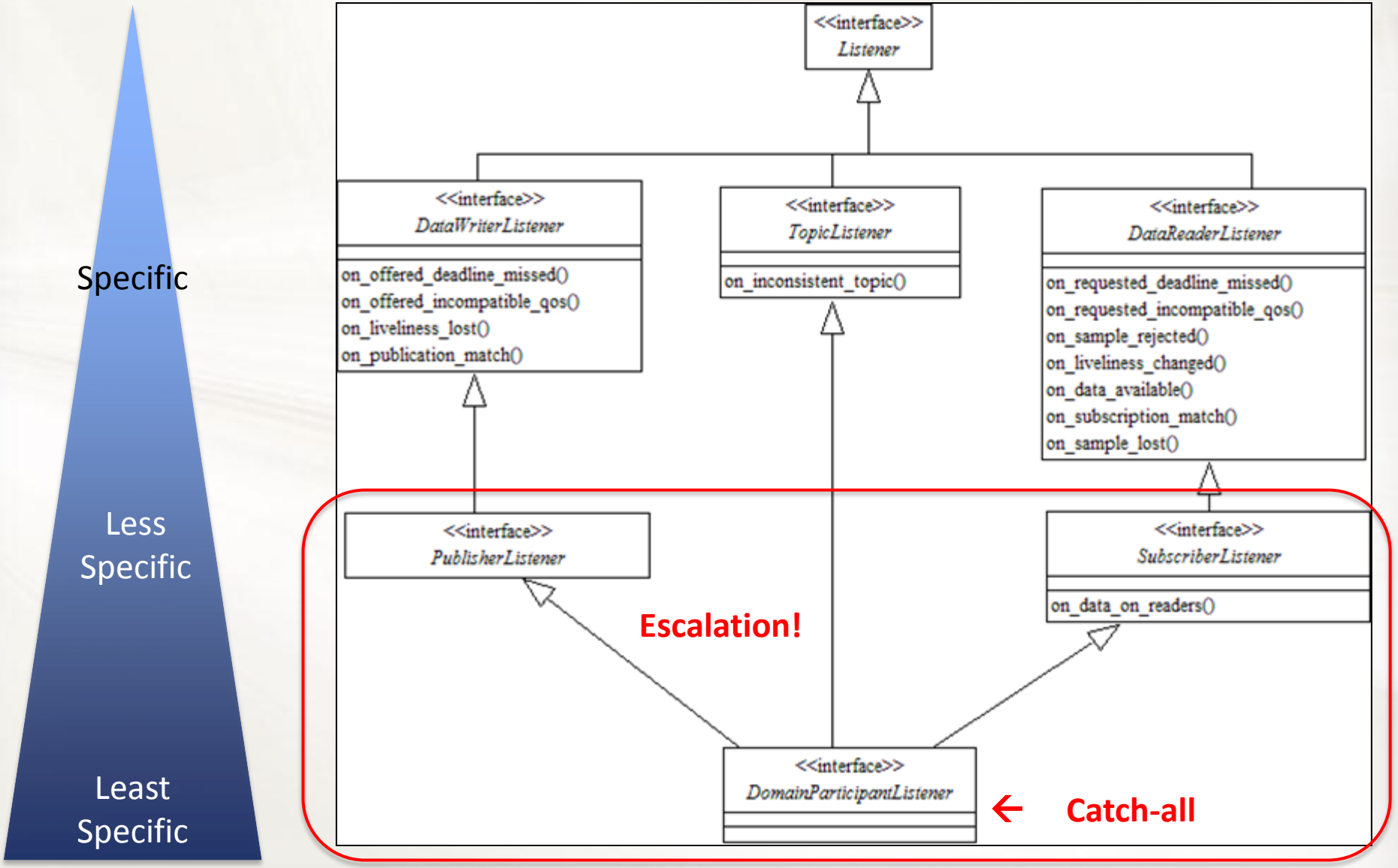
- Listeners allow asynchronous notification of entity status changes
- DataReader
  - Data available, requested deadline missed, liveliness changed, requested incompatible QoS, sample rejected, sample lost, subscription matched
- DataWriter
  - Offered deadline missed, liveliness lost, offered incompatible qos, publication matched

Entity-Specific Listener Callbacks

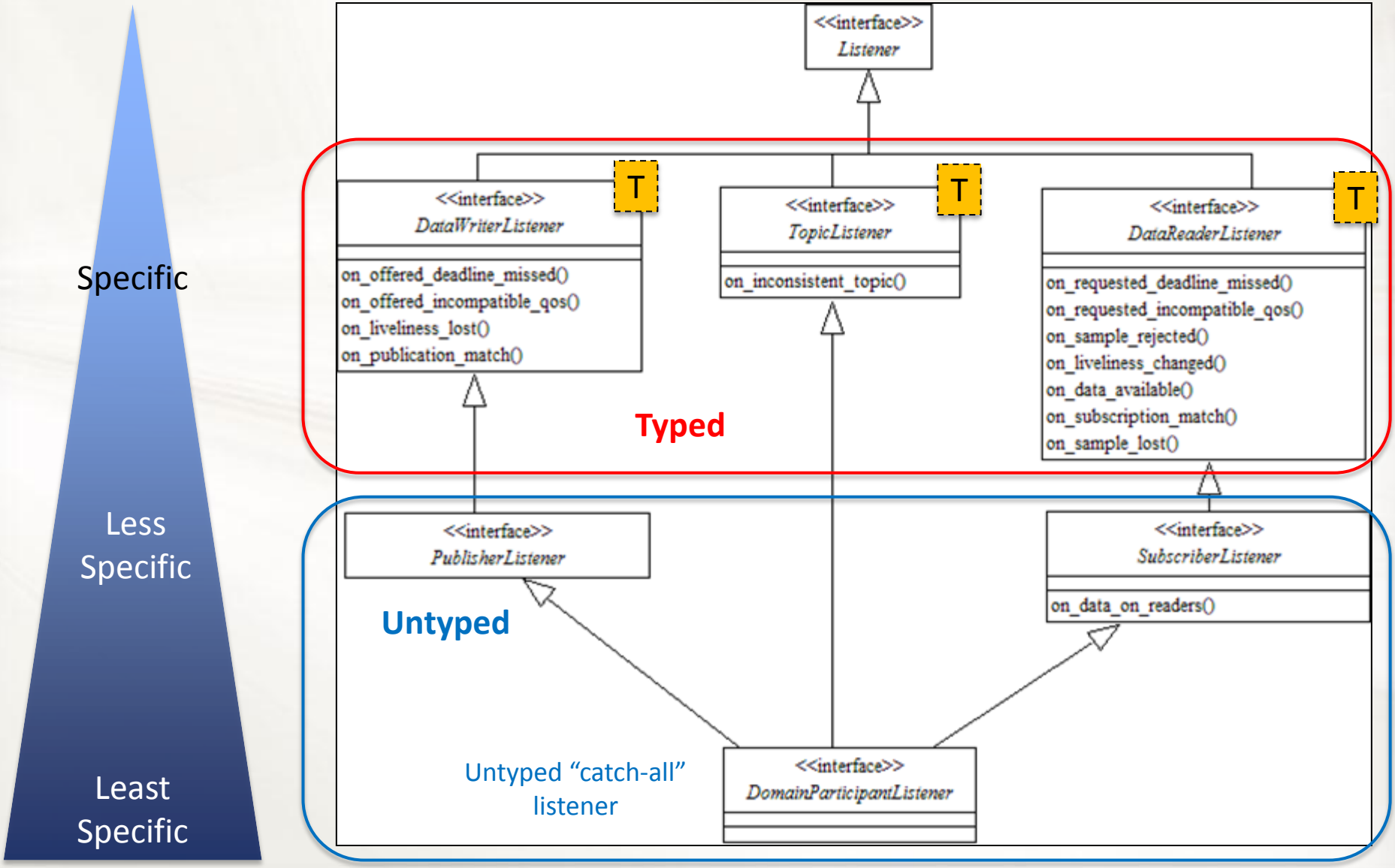
Specific



“Escalation” of Listener Callbacks



“Escalation” of Listener Callbacks



Typed Listeners

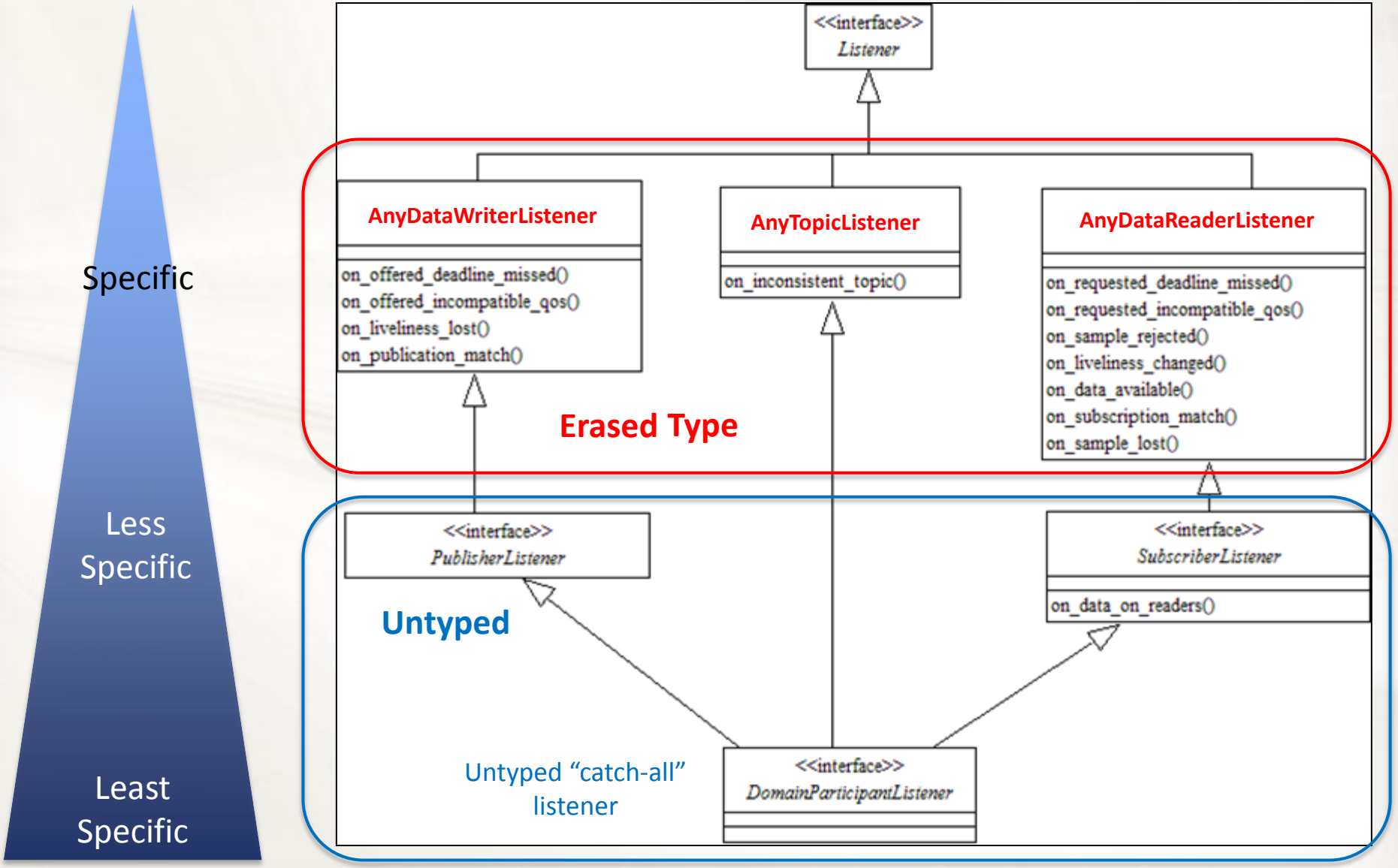
```
template <typename T>
class dds::sub::DataReaderListener {
public:
  virtual ~DataReaderListener() {}

  virtual void on_requested_deadline_missed(
    DataReader<T>& the_reader,
    const RequestedDeadlineMissedStatus&) = 0;

  virtual void on_requested_incompatible_qos(
    DataReader<T>& the_reader,
    const RequestedIncompatibleQosStatus&) = 0;

  virtual void on_sample_rejected(
    DataReader<T>& the_reader,
    const SampleRejectedStatus&) = 0;

  virtual void on_liveliness_changed(
    DataReader<T>& the_reader,
    const LivelinessChangedStatus&) = 0;

  virtual void on_data_available(
    DataReader<T>& the_reader) = 0;

  virtual void on_subscription_matched(
    DataReader<T>& the_reader,
    const SubscriptionMatchedStatus&) = 0;

  virtual void on_sample_lost(
    DataReader<T>& the_reader,
    const SampleLostStatus&) = 0;
};
```

```
template <typename T>
class dds::pub::DataWriterListener {
public:
  virtual ~DataWriterListener() { }

  virtual void on_offered_deadline_missed(
    DataWriter<T>& writer,
    const OfferedDeadlineMissedStatus &)=0;

  virtual void on_offered_incompatible_qos(
    DataWriter<T> writer,
    const OfferedIncompatibleQosStatus &)=0;

  virtual void on_liveliness_lost(
    DataWriter<T>& writer,
    const LivelinessLostStatus &)=0;

  virtual void on_publication_matched(
    DataWriter<T>& writer,
    const PublicationMatchedStatus &)=0;
};
```

Inheritance Dilemma

- Untyped "catch-all" listeners inherit from typed listeners
  - E.g., `DomainParticipantListener`
  - One place to handle status changes for all the constituent Publishers, Subscribers, DataReaders, DataWriters.
- However, untyped listeners have no knowledge of type T
  - Type T is a purely DataReader/DataWriter concern
  - Publishers, Subscribers, and DomainPartcipant don't have much to do with the type
    - Except that the type must be registered with the DomainParticipant
- C++ Issue
  - How to pass a typed DataReader/DataWriter to the untyped listener which has no knowledge of type T?
  - How to "erase" type information superficially but not permanently?

Type-Erased Listeners



Type Erasure for DataReader, DataWriters, and Topics

- DataReaders, DataWriters, Topics, and TopicDescriptions are parameterized over types (E.g., `DataReader<Foo>`)
- `std::vector<DataReader>` cannot be created but would be useful
- Welcome type-erased entities
  - `AnyDataReader, AnyDataWriter, AnyTopic, AnyTopicDescription`
- Access type independent API
- Create `std::vector<AnyDataWriter>`


```
class dds::pub::AnyDataWriter {
public:
  const dds::pub::qos::DataWriterQos& qos() const;
  void qos(const ::dds::pub::qos::DataWriterQos& q);
  const std::string& topic_name() const;
  const std::string& type_name() const;
  void wait_for_acknowledgments(const dds::core::Duration& timeout);
  void close();
  void retain(bool b);
  // ...
};
```

Inside AnyDataWriter (Type Erasure)

- `AnyDataWriter` constructor accepts any typed DataWriter
- Internally it is stored without type information
- `AnyDataWriter::get<T>` retrieves the typed DataWriter

```
namespace dds { namespace pub { namespace detail {
  class DWHolderBase;
  template <typename T> class DWHolder; // inherits DWHolderBase
} } }

class dds::pub::AnyDataWriter {
public:
  template <typename T>
  AnyDataWriter(const dds::pub::DataWriter<T>& dw)
    : holder_(new detail::DWHolder<T>(dw)) {}

  template <typename T>
  dds::pub::DataWriter<T> get() {
    return dynamic_cast<DWHolder<T> *>(holder_)->get();
  }

private:
  detail::DWHolderBase* holder_;
  // ...
};
```

Inside AnyDataWriter

```
class dds::pub::detail::DWHolderBase {
public:
  virtual ~DWHolderBase() = 0;
  virtual const dds::pub::qos::DataWriterQos& qos() const = 0;
  virtual void qos(const ::dds::pub::qos::DataWriterQos& qos) = 0;
  virtual const std::string& topic_name() const = 0;
  virtual const std::string& type_name() const = 0;
};

template <typename T>
class dds::pub::detail::DWHolder : public DWHolderBase  {
public:
  DWHolder(const dds::pub::DataWriter<T>& dw) : dw_(dw) { }
  virtual ~DWHolder() { }
  dds::pub::DataWriter<T> get() const { return dw_; }
  // Implement the rest of the DWHolderBase abstract base class
private:
  dds::pub::DataWriter<T> dw_;
};
```

Mapping Enumerations

- Classic DDS API uses C++ enumerations to model entity status kinds, QoS policy kinds, sample states, view states, instance states. For example,

```
enum StatusKind {
  DDS_INCONSISTENT_TOPIC_STATUS , DDS_SAMPLE_LOST_STATS, DDS_SAMPLE_REJECTED_STATUS, …
};
enum DurabilityKind {
  VOLATILE, TRANSIENT, TRANSIENT_LOCAL, PERSISTENT
};
```

- C++ enumerations are convenient because bitwise-OR allows *masking*
  - E.g., StatusMask for listeners and status conditions
- C++ enumerations, however, suffer from two major problems
  - Implicit conversion to integer, long, etc.
  - Scoping issues

```
StatusKind status = INCONSISTENT_TOPIC;
bool flag = (status > PERSISTENT);   // oops!

DataReader<Foo> dr(sub, topic,qos,listener, DDS_SAMPLE_LOST_STATUS | TRANSIENT);
// oops!
```

Improving Type-Safety

- DDS-PSM-Cxx improves type-safety
  - `SampleState`, `ViewState`, `InstanceState` are classes with *named constructors*
  - `DataState` and `StatusMask` are classes with named constructors (and possibly fluid interfaces)
  - Policy kinds are represented using the *type-safe enumeration idiom*

```
class dds::sub::status::SampleState
  : public std::bitset<BIT_COUNT>
{
 static const SampleState read();
 static const SampleState not_read();
 static const SampleState any();
}
```

```
class StatusMask
  : public std::bitset<STATUS_COUNT>
{
  static const StatusMask inconsistent_topic();
  static const StatusMask sample_lost();
  static const StatusMask sample_rejected();
  // many more
};
```

```
// For example
DataReader<Track> datareader (sub, topic, qos, listener,
                              StatusMask::sample_lost().inconsistent_topic().sample_rejected());

datareader.Select().state(DataState(SampleState::not_read(),
                                    ViewState::new_view(),
                                    InstanceState::alive())).take();
```

Safer Enumerations

```
int main (void)
{
  DurabilityKind status = DurabilityKind::VOLATILE;
  bool flag = (status > PERSISTENT);   // Compiler error
}
```

```
int main (void)
{
  DurabilityKind status = DurabilityKind::VOLATILE;
  bool flag = (status > DurabilityKind::PERSISTENT);   // fine!
}
```

PolicyKinds using the Type-safe Enumerations

```
template <typename def,
          typename inner = typename def::type>
class dds::core::safe_enum : public def    {
  typedef typename def::type type;
  inner val;
public:
  safe_enum(type v) : val(v) {}
  inner underlying() const { return val; }
  bool operator == (const safe_enum & s) const;
  bool operator != (const safe_enum & s) const;
  // other operators
};

namespace dds { namespace core { namespace policy {
  struct History_def {
     enum type { KEEP_LAST, KEEP_ALL };
  };
  typedef dds::core::safe_enum<History_def> HistoryKind;

  struct Durability_def {
    enum type { VOLATILE, TRANSIENT_LOCAL,
                TRANSIENT, PERSISTENT };
  };
  typedef dds::core::safe_enum<Durability_def> DurabilityKind;
} } }
```

```
namespace dds {
  namespace core {
    namespace policy {

template <class DELEGATE>
class History : public Value<DELEGATE> {
public:
  History();
  History(HistoryKind kind, int32_t depth);

  HistoryKind kind() const;
  History & kind(HistoryKind kind);
  // ...
};

}
}
}
```

Exception-Safety: Read/Take API

- 6 basic functions

```
template <typename T, template <typename Q> class DELEGATE>
class dds::sub::DataReader
{
  LoanedSamples<T> read(); // LoanedSamples<T> provides begin/end
  LoanedSamples<T> take();

  template <typename SamplesFWIterator>
  uint32_t read(SamplesFWIterator sfit, uint32_t max_samples); // forward iterator

  template <typename SamplesFWIterator>
  uint32_t take(SamplesFWIterator sfit, uint32_t max_samples); // forward iterator

  template <typename SamplesBIIterator>
  uint32_t read(SamplesBIIterator sbit); // back-insert iterator

  template <typename SamplesBIIterator>
  uint32_t read(SamplesBIIterator sbit); // back-insert iterator
};
```

Is this API Exception-Safe?

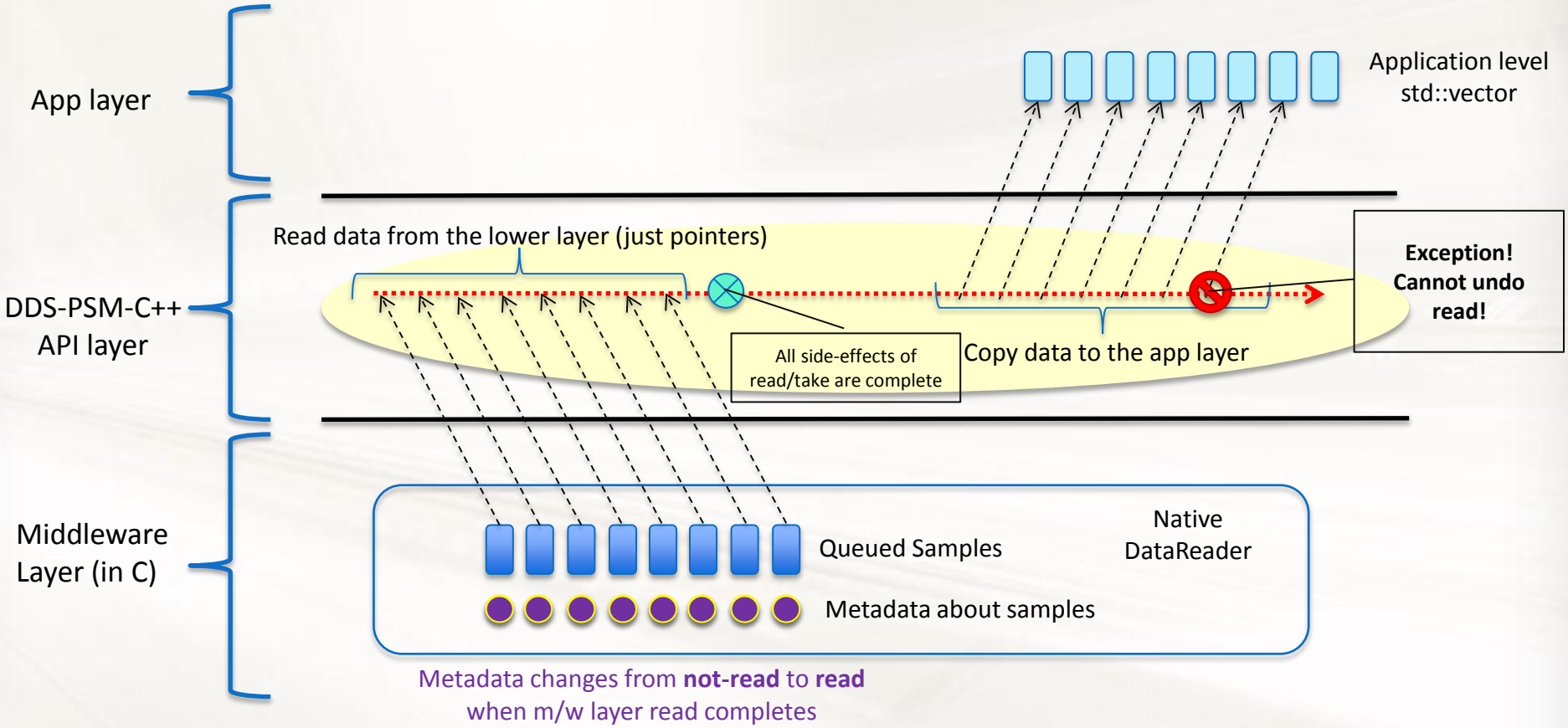
Exception Safety: Read/Take API

- Exception-Safety Recap
  - **Basic:** No resource leaks
  - **Strong:** Object state remains unaltered. *Roll-back* semantics
  - **No-throw:** The operation will not throw—ever
  - **Exception Neutrality:** Propagate the (potentially unknown) exception as it is while maintaining basic or strong guarantee
- Consider the following function that may throw

```
template <typename SamplesBIIterator>
uint32_t DataReader<T>::read(SamplesBIIterator sbit); // back-insert iterator
```

  - The sample copy assignment operator may throw (although standard allows the pimpl idiom)
  - The `std::vector<Foo>::push_back` may throw
- Only basic guarantee
  - The application-level std::vector and sample data may have change arbitrarily in the case of exception
  - What about the middleware?
    - What is the state of the middleware?
    - Can I get the lost samples (those I could not read) again?
      - The sample that causes an exception is important for debugging
- Exception-Safety has two sides
  - Application-perspective
  - Middleware perspective

Exception-Safety Dual Perspective



- An exception in the middle C++ layer leaves data stranded
  - In practice it means data that caused exception is lost
- Lower layer API does not provide roll-back semantics—*undo* read as if nothing happened
  - Undo semantics will limit concurrency substantially
  - DDS-PSM-C++ layer and all the layers below become a critical section—No new samples can be queued, no other thread can read data, etc.

Exception-Safety: Read/Take API

- Towards a better solution
  - Do not copy data in the app-level vector
    - Consequence: Iterator-based API, although convenient, is not exception-safe
  - Simply return a container of pointers to the middleware buffers

```
Samples<T> DataReader<T>::read();
Samples<T> DataReader<T>::take();
```

    - However, the object must manage the responsibility of the m/w buffers
      - Use RAII in Samples<T> to manage the resources
      - However, copy-assign and copy-ctor of Sample<T> must now perform deep copies
        - » Making deep copies during return may itself throw!
        - » Remember: Exception-safe stack has `stack::pop` and `stack::top`.
        - » Consequence: RAII-based solution is too expensive and not exception safe

Exception-Safety: Read/Take API

- Towards a better solution
  - How about the shared pointer idiom?

```
SharedSamples<T> DataReader<T>::read()
{
  T * data = mw_read();
  std::shared_ptr sp(data, mw_custom_release);
  return SharedSamples<T>(sp);
}
```

Memory allocation! May throw

  - Creation of SharedSamples(internaly shared_ptr) may itself throw
    - shared_ptr allocates a reference count
  - Allocate memory before mw_read()

```
SharedSamples<T> DataReader<T>::read()
{
  SharedSamples<T> ss(new placeholder(), mw_custom_release);
  T * data = mw_read();
  ss->set(data);
  return ss;
}
```

Unconditional memory allocation (slow)

  - Preallocating the ref-count forces dynamic memory allocation on each read call
    - Expensive
    - There may be no data available to read/take

Exception-Safety Read/Take API

- Welcome move-semantics!
- LoanedSamples<T>

```
LoanedSamples<T> DataReader::read();
LoanedSamples<T> DataReader::take();
```

- LoanedSamples<T> moves data from within the function to outside
  - Exactly one LoanedSamples<T> object owns the data at a time
    - No copy-assign, copy-ctor. Only move-assign and move-ctor
  - Contains just a pointer to the data. Moving a pointer never throws
  - RAII still applicable: The last LoanedSamples<T> returns the buffers to the m/w
  - Can be implemented in C++03—very easy in C++11

The Move-Constructor idiom for LoanedSamples

```
template <class T>
class LoanedSamples {
    template <class U>
    struct proxy {
       U *resource_;
    };
    T * resource_;
    LoanedSamples(LoanedSamples &) throw ();
    LoanedSamples & operator = (LoanedSamples &) throw ();

public:
    explicit LoanedSamples (T * r = 0) : resource_(r) { }
    ~LoanedSamples () throw() { delete resource_; }   // Assuming std:::auto_ptr like behavior.
    LoanedSamples(proxy<T> p) throw () // The proxy move constructor
      : resource_(p.resource_)
    { }
     LoanedSamples & operator = (proxy<T> p) {
      if(this->resource_) delete resource_;
      this->resource_=  p.resource_;
      return *this;
    }
    operator proxy<T> () throw () { // A helper conversion function. Note that it is non-const
      proxy<T> p;
      p.resource_ = this->resource_;
      this->resource_ = 0;  return p;    // Resource moved to the temporary proxy object.
    }
};
template <class T>
LoanedSamples<T> move(LoanedSamples <T> & ls) throw() { // Convert explicitly to a non-const reference to rvalue
  return LoanedSamples<T>(detail::proxy<T>(ls));
}
```

Using LoanedSamples<T>

```
LoanedSamples<Track> source()
{
  LoanedSamples<Track> local(new Track());
  return move(local);
}
void sink (LoanedSamples<Track> ls)
{
  // Do something useful with ls. ls is deleted automatically at the end.
}
int main(void)
{
  LoanedSamples<Track> ls(source());   // OK
  LoanedSamples<Track> ls2;
  ls2 = ls;                            // Compiler error
  ls2 = move(ls);                      // OK
  sink(ls2);                           // Compiler error
  sink(move(ls2));                     // OK
}
```

LoanedSamples<T> and SharedSamples<T>

- LoanedSamples<T> is a *move-only* type
  - C++03 standard library does not play nicely with move-only types
  - E.g., std::vector<std::auto_ptr<T>> fails to compile
  - LoanedSamples<T> move-constructor and move-assign operators are private
  - LoanedSamples<T> resists sharing
- SharedSamples<T>
  - References counted container of Ts
  - Buffers returned to the m/w when all reference cease to exist
  - Works fine with STL because copy-assign and copy-ctor are public
  - Created from LoanedSamples<T> only
  - Extra cost of allocating the reference-count is now explicit (pay only if you use it)

DDS API for C++11

- DDS-PSM-Cxx makes special provisions for C++11 environment
  - LoanedSamples<T> is a first-class move-only type
  - Namespace-level `begin, end, cbegin, and cend` functions to facilitate C++11 range-based for loop

```
LoanedSamples<Foo> ls = datareader.read();
for(auto & sample : ls) {
  /* use sample */
}
```

  - `dds::core::array` is a template alias for `std::array`

```
namespace dds { namespace core {
  template <class T>
  using array = std::array<T>;
} }
```

Type-safe Enumerations in C++03/11

```
namespace dds { namespace core { namespace policy {
  struct History_def {
     enum type { KEEP_LAST, KEEP_ALL };
  };
  typedef dds::core::safe_enum<History_def> HistoryKind;

  struct Durability_def {
    enum type { VOLATILE, TRANSIENT_LOCAL,
                TRANSIENT, PERSISTENT };
  };
  typedef dds::core::safe_enum<Durability_def>
            DurabilityKind;
} } }
```

C++03

Syntactically compatible

E.g.,

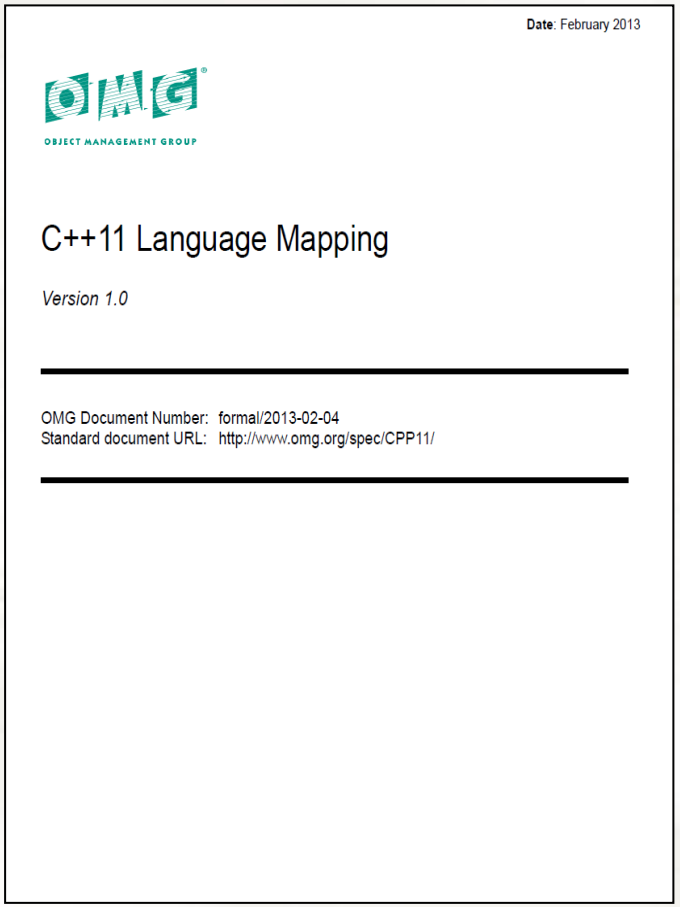
`HistoryKind::KEEP_ALL`

```
namespace dds { namespace core { namespace policy {

enum class HistoryKind { KEEP_LAST, KEEP_ALL };

enum class DurabilityKind { VOLATILE, TRANSIENT_LOCAL,
                            TRANSIENT, PERSISTENT };
} } }
```

C++11

C++11 Language Binding for User-Defined Types

- IDL to C++11 language mapping in a slide
  - OMG standard
  - http://www.omg.org/spec/CPP11/1.0/
  - DDS-PSM-Cxx uses the IDL to C++11 mapping with some minor tweaks
    - No CORBA::TypeCode, Any, Fixed types, etc.
    - Mapping of *struct* is the most relevant
  - Uses many C++11 features and libraries
    - nullptr, strongly typed enums, constexpr, move-semantics, uniform initialization, final, etc.
    - std::shared_ptr, std::weak_ptr, std::array, type traits, exceptions, etc.


Date: February 2013

OMG
OBJECT MANAGEMENT GROUP

C++11 Language Mapping

Version 1.0

OMG Document Number: formal/2013-02-04
Standard document URL: http://www.omg.org/spec/CPP11/

Mapping User-Defined Data Types in C++11

IDL

```
typedef sequence<string> Authors;

struct Book {
  string          title;
  Authors         authors;
  string          publisher;
  long            pub_year;
};
```

C++11

```
class Book {
  Book();
  Book(const Book &);
  Book(Book &&);
  Book & operator = (const Book &);
  Book & operator = (Book &&);

  Book (std::string title,
        std::vector<string> authors,
        std::string publisher,
        int32_t pub_year);
  // ...
};
```

- Mapping for IDL struct types uses the pass-by-value idiom
  - The constructor take parameters by value
  - A typical implementation will std::move the parameters
  - The spec uses move-friendly types only
  - See C++11 Idioms talk @ Silicon Valley Code Camp 2012 for more details

Thank You!

- More Information
  - DDS PSM C++
    - http://www.omg.org/spec/DDS-PSM-Cxx/
  - Real-Time Innovations
    - www.rti.com
  - More C++ Idioms
    - http://en.wikibooks.org/wiki/More_C++_Idioms