May 2010

Extensible and Dynamic Topic Types for DDS

Beta 1

OMG Adopted Specification

OMG Document Number: ptc/2010-05-12
Standard Document URL: http://www.omg.org/spec/DDS-XTypes
Associated Schema files*: http://www.omg.org/spec/DDS-XTypes/20100501
http://www.omg.org/spec/DDS-XTypes/20100502
http://www.omg.org/spec/DDS-XTypes/20100503

---

* Original files: mars/2010-02-02 (XMI), mars/2010-02-03 (XSD), mars/2010-02-04 (IDL)

This OMG document replaces the submission document (mars/2010-02-01, Alpha). It is an OMG Adopted Beta 1 specification currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by November 29, 2010.

You may view the pending issues for this specification from the OMG revision issues web page http://www.omg.org/issues/.

The FTF Recommendation and Report for this specification will be published on April 4, 2011. If you are reading this after that date, please download the related documents from the OMG Specifications Catalog.

Beta 2 represents the end product (finalized specification) of the FTF process. The Beta 2 document will be available from this URL: http://www.omg.org/spec.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

OMG's Issue Reporting Procedure

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

Table of Contents

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

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

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

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- OMG SysML™
- Other Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- DDS and the DDS Interoperability Protocol, RTPS
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications

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

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

OMG Headquarters
140 Kendrick Street
Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

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

Typographical Conventions

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

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold: Programming language elements.`

Helvetica/Arial - 10 pt: Exceptions

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

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/technology/agreement.htm.

1. Scope

The Specification addresses four related concerns summarized in the figure below.

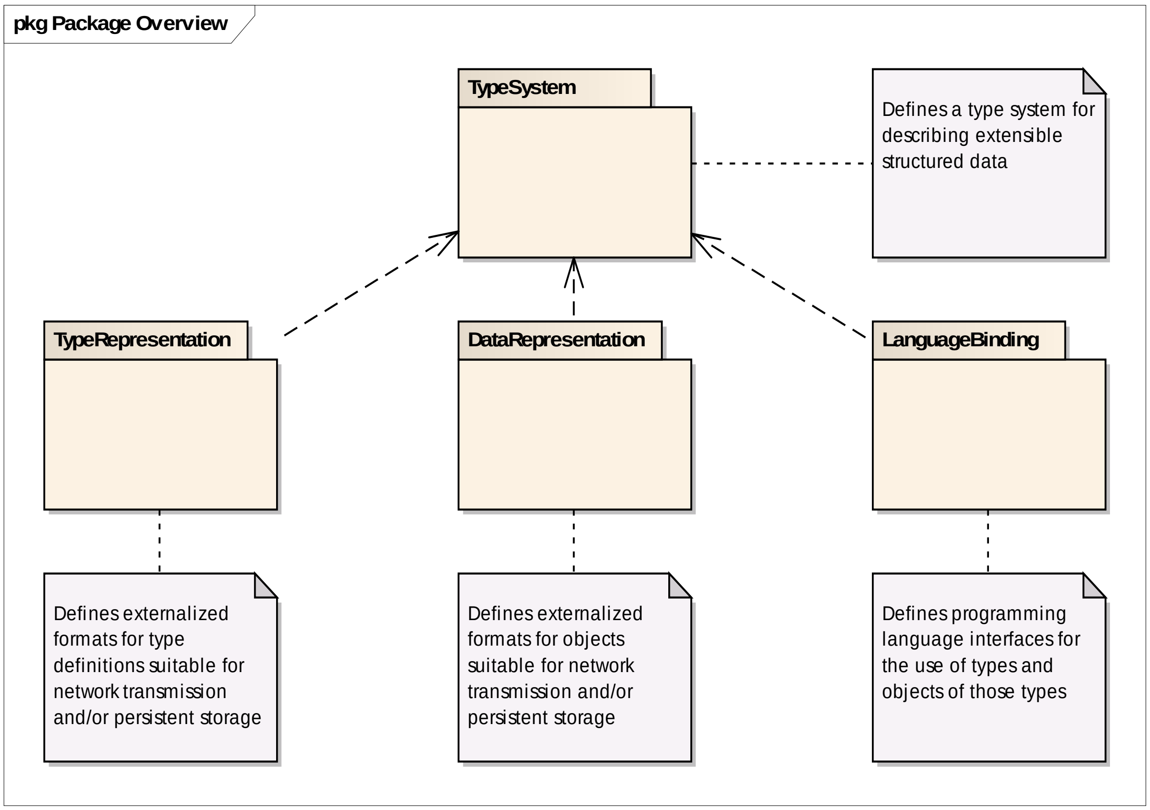

Figure 1 - Packages

> *The specification addresses four related concerns: the type system, the representation of types, the representation of data, and the language bindings used to access types and data. Each of these concerns is modeled as a collection of classes belonging to a corresponding package.*

This specification provides the following additional facilities to DDS [DDS] implementations and users:

- **Type System**. The specification defines a model of the data types that can be used for DDS Topics. The type system is formally defined using UML. The Type System is defined in section 7.2 and its subsections. The structural model of this system is defined in the Type System Model in section 7.2.2. The framework under which types can be modified over time is summarized in section 7.2.3, "Type Extensibility and Mutability." The concrete rules under which the concepts from 7.2.2 and 7.2.3 come together to define compatibility in the face of such modifications are defined in section 7.2.4, "Type Compatibility: "is-assignable-from" relationship."
- **Type Representations**. The specification defines the ways in which types described by the Type System may be externalized such that they can be stored in a file or communicated over a network. The specification adds additional Type Representations beyond the one (IDL [IDL]) already implied by the DDS specification. Several Type Representations are specified in the subsections of section 7.3. These include IDL (7.3.1), XML (7.3.2), XML Schema (XSD) (7.3.3), and TypeObject (7.3.4).

- **Data Representation**. The specification defines multiple ways in which objects of the types defined by the Type System may be externalized such that they can be stored in a file or communicated over a network. (This is also commonly referred as “data serialization” or “data marshaling.”) The specification extends and generalizes the mechanisms already defined by the DDS Interoperability specification [RTPS]. The specification includes Data Representations that support data type evolution, that is, allow a data type to change in certain well-defined ways without breaking communication. Two Data Representations are specified in the subsections of section 7.4. These are Extended CDR (7.4.1) and XML (7.4.2).
- **Language Binding**. The specification defines multiple ways in which applications can access the state of objects defined by the Type System. The submission extends and generalizes the mechanism currently implied by the DDS specification (“Plain Language Binding”) and adds a Dynamic Language Binding that allows application to access data without compile-time knowledge of its type. The specification also defines an API to define and manipulate data types programmatically. Two Language Bindings are specified in the subsections of section 7.5. These are the Plain Language Binding and the Dynamic Language Binding.

2. Conformance Criteria

This specification recognizes two levels of conformance: (*1*) conformance with respect to programming interfaces—that is, at the level of the DDS API—and (*2*) conformance with respect to network interoperability—that is, at the level of the RTPS protocol. An implementation may conform to either or both of these levels, just as it may conform to either DDS and/or RTPS.

These conformance levels are formally defined as follows. Conformance to sections of this specification not specifically identified below is required, regardless of the conformance level.

2.1 Programming Interface Conformance

This specification extends the *Data Distribution Service for Real-Time Systems* specification [DDS] with an additional optional conformance profile: the “Extensible and Dynamic Types Profile.” Conformance to this specification with respect to programming interfaces shall be equivalent to conformance to the DDS specification with respect to at least the existing Minimum Profile and the new Extensible and Dynamic Types Profile. Implementations may conform to additional DDS profiles.

The new Extensible and Dynamic Types profile of DDS shall consist of the following sections of this specification:

- Extensible and Dynamic Topic Types for DDS (Chapter 7) up to and including *Type Representation* (Section 7.3)
- *Language Binding* (Section 7.5)
- Use of the Type System by DDS (Section 7.6)
- All annexes pertaining to the above

2.2 Network Interoperability Conformance

Conformance with respect to network interoperability shall consist of conformance to the following sections of this specification:

- Extensible and Dynamic Topic Types for DDS (Chapter 7) up to and including Scope (Section 7.1)
- Representing Types with `TypeObject` (Section 7.3.4)
- *Data Representation* (Section 7.4) and XSD Type Representation (Section 7.3.3). The XML schemas defined by Section 7.3.3 in turn describe the structure of the XML documents defined in XML Data Representation – Section 7.4.2).

- Use of the Type System by DDS (Section 7.6) up to and including Discovery (Section 7.6.2).
- All annexes pertaining to the above

In addition, conformance at this level requires conformance to the *Real-Time Publish-Subscribe Wire Protocol* specification [RTPS].

3. Normative References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this specification.

- [**DDS**] *Data Distribution Service for Real-Time Systems Specification*, Version 1.2 (OMG document formal/2007-01-01)
- [**RTPS**] *Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification*, Version 2.1 (OMG document formal/2009-01-05)
- [**IDL**] *Common Object Request Broker Architecture (CORBA) Specification*, Version 3.1, Part 1 (OMG document formal/2008-01-04), section 7: "OMG IDL Syntax and Semantics"
- [**CDR**] *Common Object Request Broker Architecture (CORBA) Specification*, Version 3.1, Part 2 (OMG document formal/2008-01-07), section 9.3: "CDR Transfer Syntax"
- [**C-LANG**] *Programming languages -- C* (ISO/IEC document 9899:1990)
- [**C++-LANG**] *Programming languages -- C++* (ISO/IEC document 14882:2003)
- [**JAVA-LANG**] *The Java Language Specification, Second Edition* (by Sun Microsystems, http://java.sun.com/docs/books/jls/)
- **[C-MAP**] *C Language Mapping Specification*, Version 1.0 (OMG document formal/1999-07-35)
- [**C++-MAP**] *C++ Language Mapping Specification*, Version 1.2 (OMG document formal/2008-01-09)
- [**JAVA-MAP**] *IDL to Java Language Mapping*, Version 1.3 (OMG document formal/2008-01-11)
- [**IDL-XSD**] *CORBA to WSDL/SOAP Interworking Specification*, Version 1.2.1 (OMG document formal/2008-08-03)
- [**LATIN**] *Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1* (ISO/IEC document 8859-1:1998)
- [**UCS**] *Information technology -- Universal Multiple-Octet Coded Character Set (UCS)* (ISO/IEC document 10646:2003)
- **[FNMATCH]** *POSIX fnmatch function* (IEEE 1003.2-1992 section B.6)

4. Terms and Definitions

***Data Centric Publish-Subscribe (DCPS)* –** The mandatory portion of the DDS specification used to provide the functionality required for an application to publish and subscribe to the values of data objects.

Data Distribution Service (DDS) – An OMG distributed data communications specification that allows Quality of Service policies to be specified for data timeliness and reliability. It is independent of implementation languages.

5. Symbols

No additional symbols are used in this specification.

6. Additional Information

6.1 Data Distribution Service for Real-Time Systems (DDS)

The *Data Distribution Service for Real-Time Systems* (DDS) is the Object Management Group (OMG) standard for data-centric publish-subscribe communication. This standard has experienced a record-pace adoption within the Aerospace and Defense domain and is swiftly expanding to new domains, such as Transportation, Financial Services, and SCADA. To sustain and further propel its adoption, it is essential to extend the DDS standard to effectively support a broad set of use cases.

The OMG DDS specification has been designed to effectively support statically defined data models. This assumption requires that the data types used by DDS Topics are known at compile time and that every member of the DDS global data space *agrees* precisely on the same topic-type association. This model allows for good properties such as static type checking and very efficient, low-overhead, implementation of the standard. However it also suffers a few drawbacks:

- It is hard to cope with data models evolving over time unless all the elements of the system affected by that change are upgraded consistently. For example, the addition or removal of a field in the data type it is not possible unless all the components in the system that use that data type are upgraded with the new type.
- Applications using a data type must know the details of the data type at compile time, preventing use cases that would require dynamic discovery of the data types and their manipulation without compile-time knowledge. For example, a data-visualization tool cannot discover dynamically the type of a particular topic and extract the data for presentation in an interface.

With the increasing adoption of DDS for the integration of large distributed systems, it is desirable to provide a mechanism that supports evolving the data types without requiring all components using that type to be upgraded simultaneously. Moreover it is also desirable to provide a "dynamic" API that allows type definition, as well as publication and subscription data types without compile-time knowledge of the schema.

Most of the concerns outlined in Scope above (Type System, Type Representation, etc.) are already addressed in the DDS specification and/or in the DDS Interoperability Protocol specification. However, these specifications sometimes are not sufficiently explicit, complete, or flexible with regards to the above concerns of large dynamic systems. This specification addresses those limitations.

The current mechanisms used by the existing specifications are shown in the table below.

Table 1 – Type-related concerns addressed by this specification

Concern	***Mechanism currently in use by DDS and the Interoperability Protocol***
Type System	The set of "basic" IDL types: primitive types, structures, unions, sequences, and arrays. This set is only implicitly defined.
Type Representation	Uses OMG Interface Definition language (IDL). This format is used to describe types on a file. There is no representation provided for communication of types over the network.
Data Representation	The DDS Interoperability Protocol uses the OMG Common Data Representation (CDR) based on the corresponding IDL type. It also uses a "parameterized" CDR representation for the built-in Topics, which supports schema evolution.
Language Binding	Plain Language objects as defined by the IDL-to-language mapping.

This specification formally addresses each of the aforementioned concerns and specifies multiple mechanisms to address each concern. Multiple mechanisms are required to accommodate a broad range of application requirements and balance tradeoffs such as efficiency, evolvability, ease of integration with other technologies (such as Web Services), as well as compatibility with deployed systems. Care has been taken such that the introduction of multiple mechanisms does not break existing systems nor make it harder to develop future interoperable systems.

Table 2 summarizes the main features and mechanisms provided by the specification to address each of the above concerns.

Table 2 – Main features and mechanisms provided by this Specification to address type-related concerns

Concern	***Features and mechanisms introduced by the extensible Topics submission***
Type System	Defined in UML, independent of any programming language. Supports: • Most of the IDL data types • Specification of additional DDS-specific concepts, such as keys • Single Inheritance • Type versioning and evolution • Sparse types (types, the samples of which may omit values for certain fields; see below for a formal treatment)
Type Representation	Several specified: • **IDL** – Supports CORBA integration and existing IDL-defined types. • **XSD –** Allows reuse of schemas defined for other purposes (e.g., in WSDL files). • **XML** – Provides a compact, XML-based representation suitable for human input and tool use.

	• `TypeObject` – The most compact representation (typically binary). Optimized for network propagation of types.
Data Representation	Several specified: • **CDR** – Most compact representation. Binary. Interoperates with existing systems. Does not support evolution. • **Parameterized CDR** – Binary representation that supports evolution. It is the most compact representation that can support type evolution. • **XML** – Human-readable representation that supports evolution.
Language Binding	Several Specified: • **Plain Language Binding** – Equivalent to the type definitions generated by existing standard IDL-to-programming language mappings. Convenient. Requires compile-type knowledge of the type. • **Dynamic Language Binding –** Allows dynamic type definition and introspection. Allows manipulation of data without compile-time knowledge.

6.2 Acknowledgments

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

- Real-Time Innovations
- PrismTech Corp
- THALES

7. Extensible and Dynamic Topic Types for DDS

7.1 Overview

A running DDS [DDS] application that publishes and subscribes data must deal directly or indirectly with data types and data samples of those types and the various representations of those objects. The application and middleware perspectives related to data and data types are shown in the figure below.

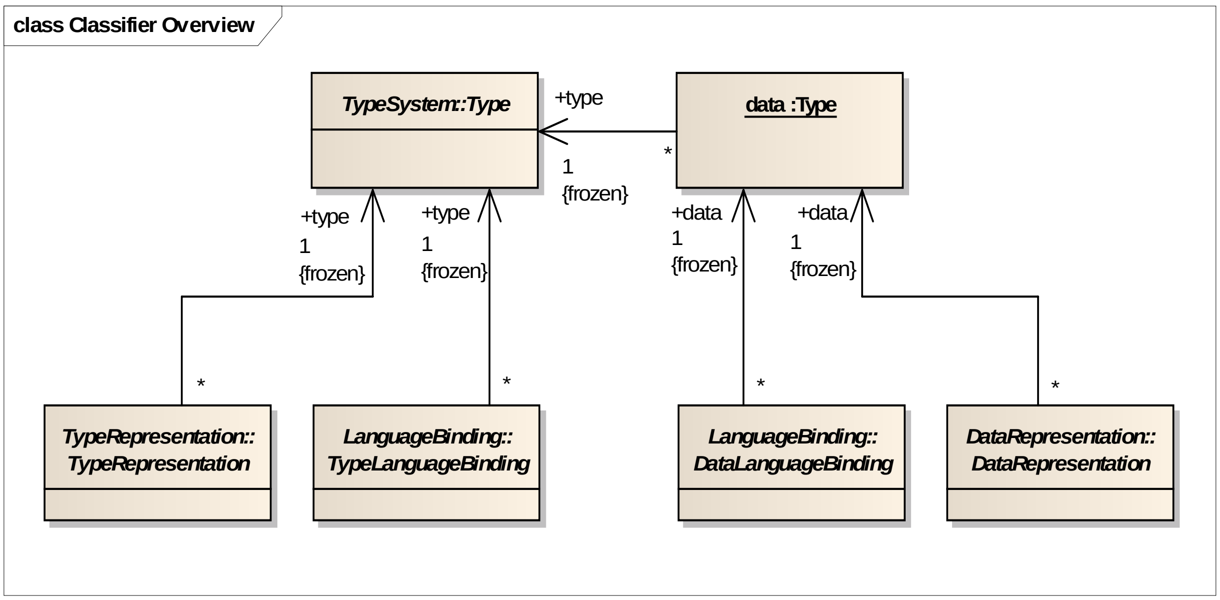

Figure 2 - Relationships between Type System, Type Representation, Language Binding and Data Representation

> *DDS data objects have an associated data type (in the common programming language sense of the word) that defines a common structure for all objects of the type. From a programming perspective, an object is manipulated using a Language Binding suitable for the programming language in use (e.g., Java). From a network communications and file storage perspective, an object must have a representation (encoding) that is platform neutral and maps into a contiguous set of bytes, whether textual or binary.*
>
> *Similarly, from a programming perspective a data type is manipulated using a Language Binding to the programming language of choice (sometimes known as a reflection API) and must have a representation (encoding) that is platform neutral and maps into a contiguous set of bytes (e.g., XSD or IDL).*

The following example is based on a hypothetical “Alarm” data use case can be used to explain the figure above.

An application concerned with alarms might use a type called “`AlarmType`” to indicate the nature of the alarm, point of origin, time when it occurred, severity etc. Applications publishing and subscribing to `AlarmType` must therefore understand to some extent the logical or semantic contents of that type. This is what is represented by the `TypeSystem::Type` class in the figure above.

If this type is to be communicated in a design document or electronically to a tool, it must be represented in some “external” format suitable for storing in a file or on a network packet. This aspect is represented by the `TypeRepresentation::TypeRepresentation` class in the figure above. A realization of the `TypeRepresentation` class may use XML, XSD, or IDL to represent the type.

An application wishing to understand the structure of the Type, or the middleware attempting to check type-compatibility between writers and readers, must use some programming language construct to examine the type. This is represented by the

`LanguageBinding::TypeLanguageBinding` class. As an example of this concept, the class `java.lang.Class` plays this role within the Java platform.

An application publishing Alarms or receiving Alarms must use some programming language construct to set the value of the alarm or access those values when it receives the data. This programming language construct may be a plain language object (such as the one generated from an IDL description of the type) or a dynamic container that allows setting and getting named fields, or some other programming language object. This is represented by the `LanguageBinding::DataLanguageBinding` class.

An application wishing to store Alarms on a file or the middleware wishing to send Alarms on a network packet or create Alarm objects from data received on the network must use some mechanism to "serialize" the Alarm into bytes in a platform-neutral fashion. This is represented by the `DataRepresentation::DataRepresentation` class. An example of this would be to use the CDR Data Representation derived from the IDL Type Representation.

The classes in the figure above represent each of the independent concerns that both application and middleware need to address. The non-normative figure below indicates their relationships to one another in a less formal way:

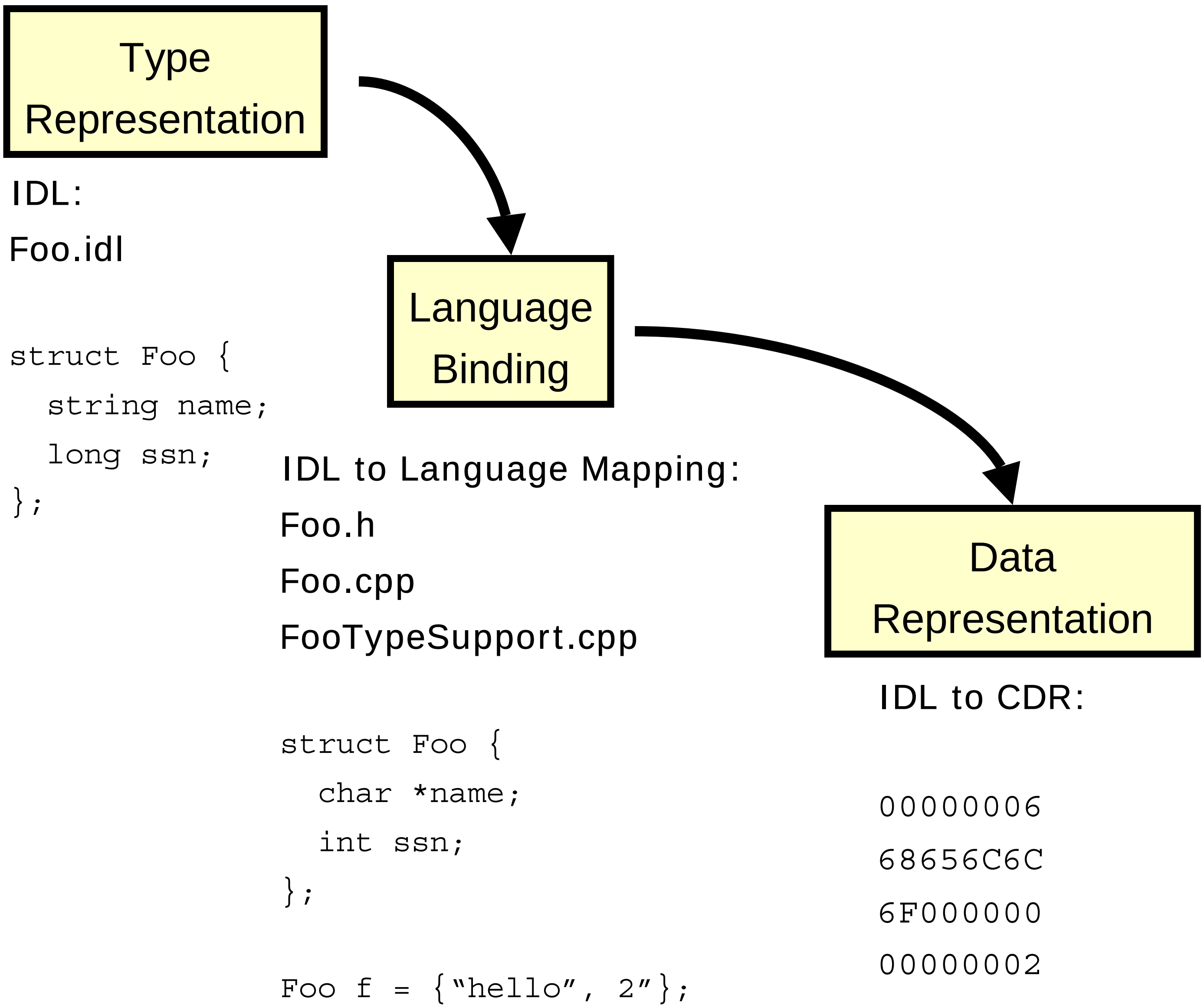

Figure 3 - Example Type Representation, Language Binding, and Data Representation

Type Representation is concerned with expressing the type in a manner suitable for human input and output, file storage, or network communications. IDL is an example of a standard type representation. Language Binding is concerned with the programming language constructs used to interact with data of a type or to introspect the type. Plain language objects as obtained from the IDL-to-language mappings of the IDL representation of the type are one possible Language Binding. Data Representation is concerned with expressing the data in a way that can be stored in a file or communicated over a network or manipulated by a human. The Common Data Representation is a Data Representation optimized for network communications; XML is another representation more suitable for human manipulation.

7.2 Type System

The Type System defines the data types that can be used for DDS Topics and therefore the type of the data that can be published and subscribed via DDS.

7.2.1 Background (Non-Normative)

The specified type system is designed to be sufficiently rich to encompass the needs of modern distributed applications and cover the basic data types available both in common programming languages such as C/C++, Java, and C#, as well as in distributed computing data-definition languages such as IDL or XDR.

The specified type system supports the following primitive types:

- Boolean type
- Byte type
- Integral types of various bit lengths (16, 32, 64), both signed and unsigned
- Floating point types of various precisions: single precision, double precision, and quad precision
- Single-byte and wide character types

In addition the specified type system covers the following non-basic types constructed as collections or aggregations of other types:

- Structures, which can singly inherit from other structures
- Unions
- Single- and multi-dimensional arrays
- Variable-length sequences of a parameterized element type
- Strings of single-byte and wide characters
- Variable-length maps of parameterized key and value types

The specified type-system supports type evolution, type inheritance, and sparse types. These concepts are described informally in Sections 7.2.1.1, 7.2.1.2, and 7.2.1.3 below and formally in Section 7.2.2.

7.2.1.1 Type Evolution Example

Assume a DDS-based distributed application has been developed that uses the Topic "Vehicle Location" of type `VehicleLocationType`. The type `VehiclePositionType` itself was defined using the following IDL:

```
// Initial Version
struct VehicleLocationType {
   float latitude;
   float longitude;
};
```

As the system evolves it is deemed useful to add additional information to the `VehicleLocationType` such as the estimated error latitude and longitude errors as well as the direction and speed resulting in:

```
// New version
struct VehicleLocationType {
   float latitude;
   float longitude;
   float latitude_error_estimate;    // added field
   float longitude_error_estimate;   // added field
   float direction;                  // added field
   float speed;                      // added field
};
```

This additional information can be used by the components that understand it to implement more elaborate algorithms that estimate the position of the vehicle between updates. However, not all components that publish or subscribe data of this type will be upgraded to this new definition of `VehicleLocationType` (or if they will not be upgraded, they will not be upgraded at the same time) so the system needs to function even if different components use different versions of `VehicleLocationType`.

The Type System supports type evolution so that it is possible to "evolve the type" as described above and retain interoperability between components that use different versions of the type such that:

- A publisher writing the "initial version" of `VehicleLocationType` will be able to communicate with a subscriber expecting the "new version" of the `VehicleLocationType`. In practice what this means is that the subscriber expecting the "new version" of the `VehicleLocationType` will, depending on the details of how the type was defined, either be supplied some default values for the added fields or else be told that those fields were not present.
- A publisher writing the "new version" of `VehicleLocationType` will be able to communicate with a subscriber reading the "initial version" of the `VehicleLocationType`. In practice this means the subscriber expecting the "initial version" of the `VehicleLocationType` will receive data that strips out the added fields.

Evolving a type requires that the designer of the new type explicitly tags the new type as equivalent to, or an extension of, the original type and limits the modifications of the type to the supported set. The addition of new fields is one way in which a type can be evolved. The complete list of allowed transformations is described in Section 7.2.4.

7.2.1.2 Type Inheritance Example

Building upon the same example in Section 7.2.1.1, assume that the system that was originally intended to only monitor location of land/sea-surface vehicles is now extended to also monitor air vehicles. The location of an air vehicle requires knowing the altitude as well. Therefore the type is extended with this field.

```
// Extended Location
struct VehicleLocation3DType : VehicleLocationType {
   float altitude;
   float vertical_speed;
};
```

VehicleLocation3DType is an extension of `VehicleLocationType`, not an evolution. VehicleLocation3DType represents a new type that extends `VehicleLocationType` in the object-oriented programming sense (IS-A relationship).

The Type System supports type inheritance so that it is possible to “extend the type” as described above and retain interoperability between components that use different versions of the type. So that:

- An application subscribing to Topic “Vehicle Position” and expecting to read `VehicleLocationType` CAN receive data from a Publisher that is writing a VehicleLocation3DType. In other words applications can write extended types and read base types.
- An application subscribing to Topic “Vehicle Position” and expecting to read VehicleLocation3DType CAN receive data from a Publisher that is writing a `VehicleLocationType`. Applications expecting the derived (extended) type can accept the base type; additional members in the derived type will take no value or a default value, depending on their definitions.

This behavior matches the behavior of the “IS-A” relationship in Object-Oriented Languages,

Intuitively this means that a VehicleLocation3DType is a new type that happens to extend the previous type. It can be substituted in places that expect a `VehiclePosition` but is not fully equivalent. The substitution only works one way: An application expecting a VehicleLocation3DType cannot accept a `VehiclePosition` in place because it is cannot “just” assume some default value for the additional fields. Rather it wants to just read those `VehiclePosition` that corresponds to Air vehicles.

7.2.1.3 Sparse Types Example

Suppose that an application publishes a stream of events. There are many kinds of events that could occur in the system, but they share a good deal of data, they must all be propagated with the same QoS, and the relative order among them must be preserved—it is therefore desirable to publish all kinds of events on a single topic. However, there are fields that only make sense for certain kinds of event. In its local programming language (say, C++ or Java), the application can assign a pointer to null to omit a value for these fields. It is desirable to extend this concept to the network and allow the application to omit irrelevant data in order to preserve the correct semantics of the data.

Alternatively, suppose that an application subscribes to data of a type containing many fields, most of which often take a pre-specified “default value” but may, on occasion, deviate from that default. In this situation it would be inefficient to send every field along with every sample. Rather it would be better to just send the fields that take a non-default value and fill the missing fields on the receiving side, or even let the receiving application do that job. This situation occurs, for example, in the DDS Built-in Topic Data. It also occurs in financial applications that use the FIX encoding for the data.

The type system supports sparse types whereby a type can have fields marked “optional” so that a Data Representation may omit those fields. Values for non-optional fields may also be omitted to save network bandwidth, in which case the Service will automatically fill in default values on behalf of the application.

7.2.2 Type System Model

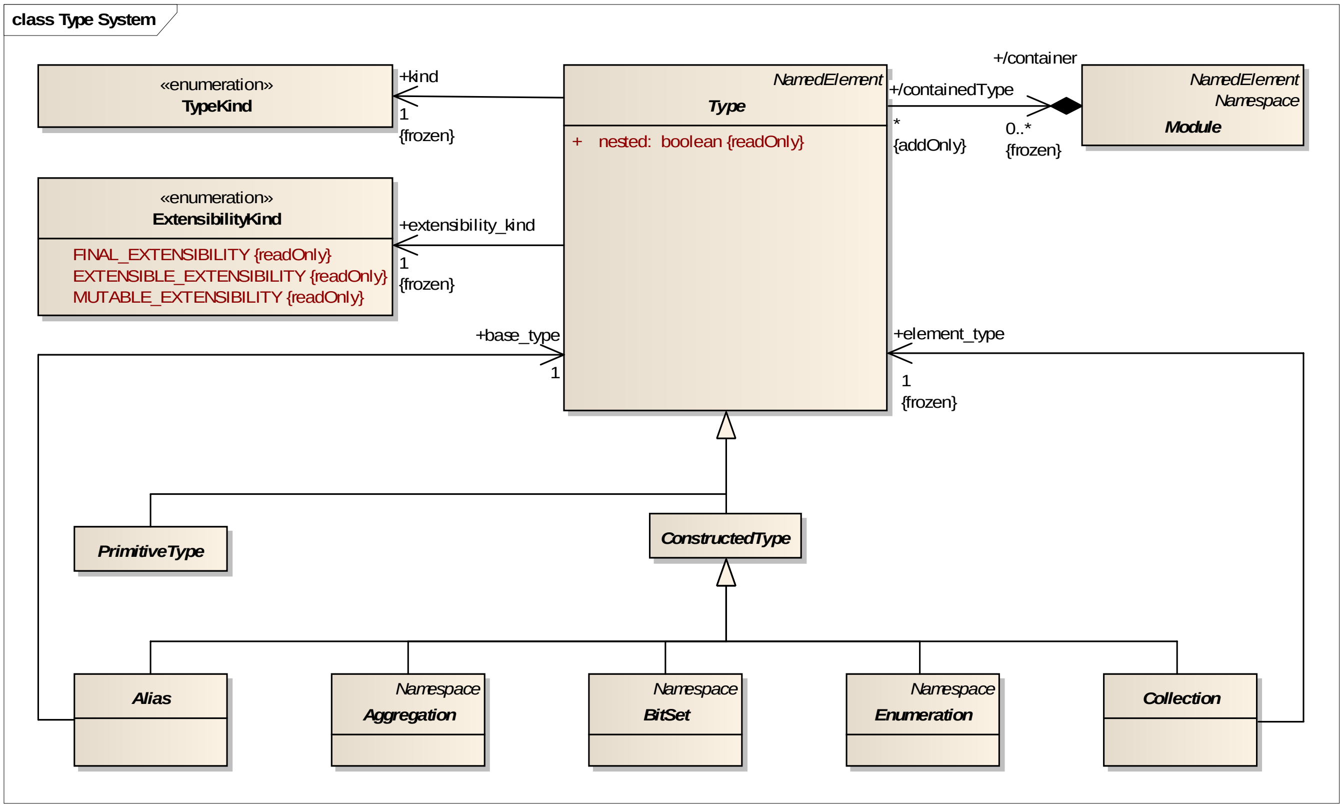

Figure 4 - Type System Model

The definition of a type in the Type System can either be primitive or it can be constructed from the definitions of other types.

The Type System model is shown in Figure 4. This model has the following characteristics:

- A type has a non-empty *name* that is unique within its namespace (see Section 7.2.2.1). The set of valid names is the set of valid identifiers defined by the OMG IDL specification [IDL].
- A type has a *kind* that identifies which primitive type it is or, if it is a constructed type, whether it is a structure, union, sequence, etc.
- The type system supports Primitive Types (i.e., their definitions do not depend on those of any other types) whose names are predefined. The Primitive Types are described in 7.2.2.2.
- The type system supports Constructed Types whose names are explicitly provided as part of the type-definition process. Constructed Types include enumerations, collections, structure, etc. Constructed types are described in Section 7.2.2.3.

7.2.2.1 Namespaces

A namespace defines the scope within which a given name must be unique. That is, it is an error for different elements within the same namespace to have the same name. However, it is legal for different elements within different namespaces to have the same name.

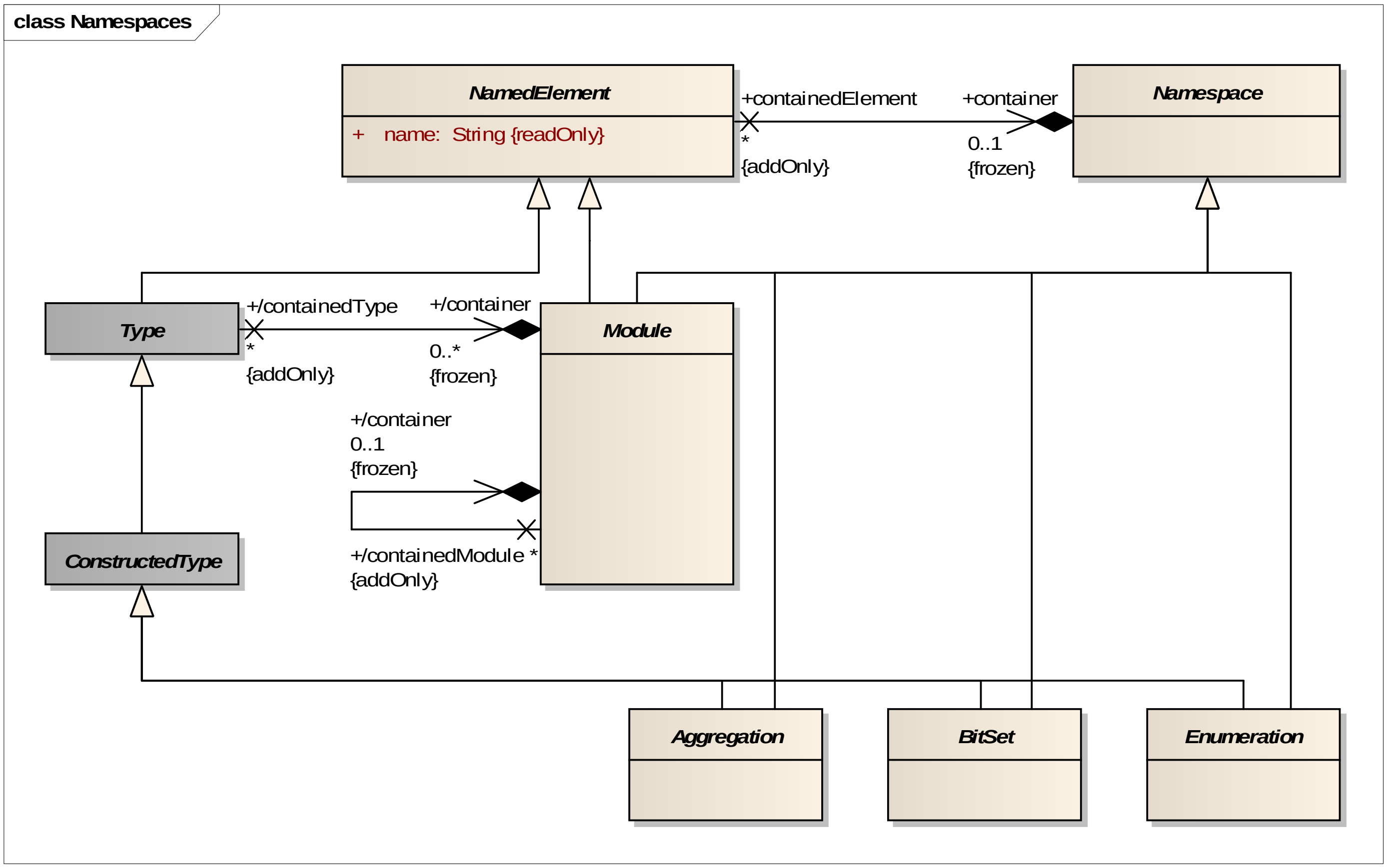

Figure 5 - Namespaces

Namespaces fall into one of two categories:

- *Modules* are namespaces whose contained named elements are types. The concatenation of module names with the name of a type inside of those modules is referred to as the type's "fully qualified name."
- Certain kinds of *types* are themselves namespaces with respect to the elements inside of them.

7.2.2.2 Primitive Types

The primitive types in the Type System have parallels in most computer programming languages and are the building blocks for more complex types built recursively as collections or aggregations of more basic types.

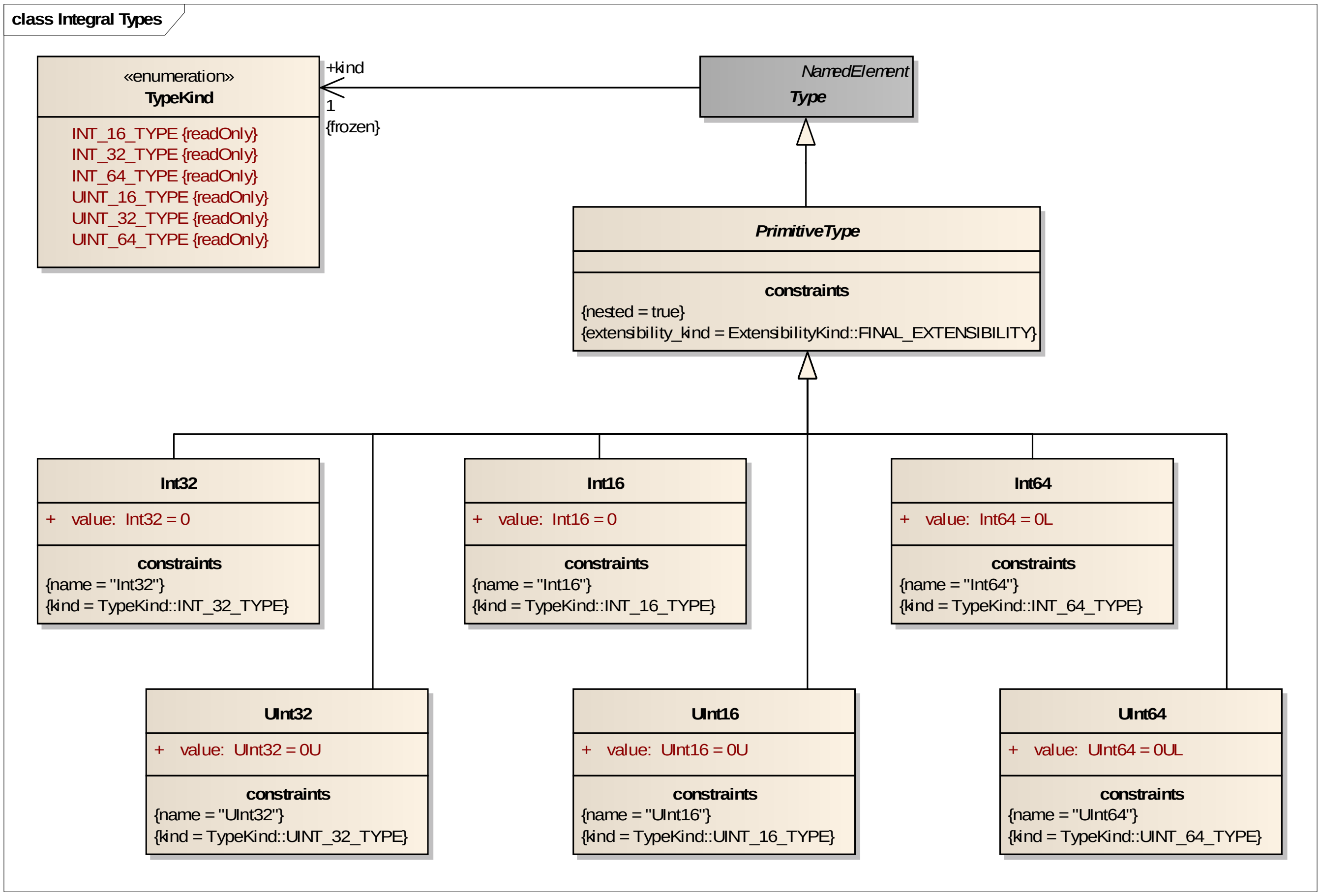

Figure 6 - Primitive Types: Integral Types

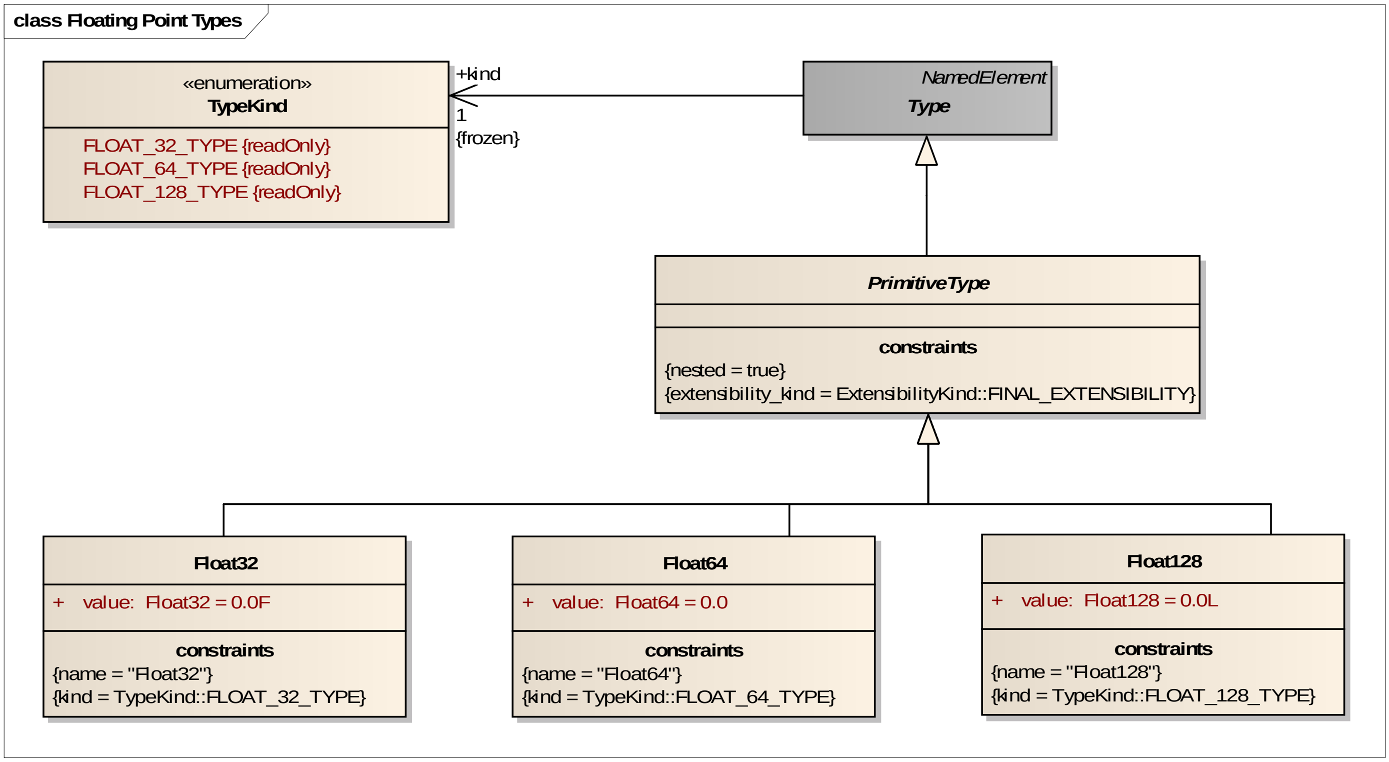

Figure 7 - Primitive Types: Floating Point Types

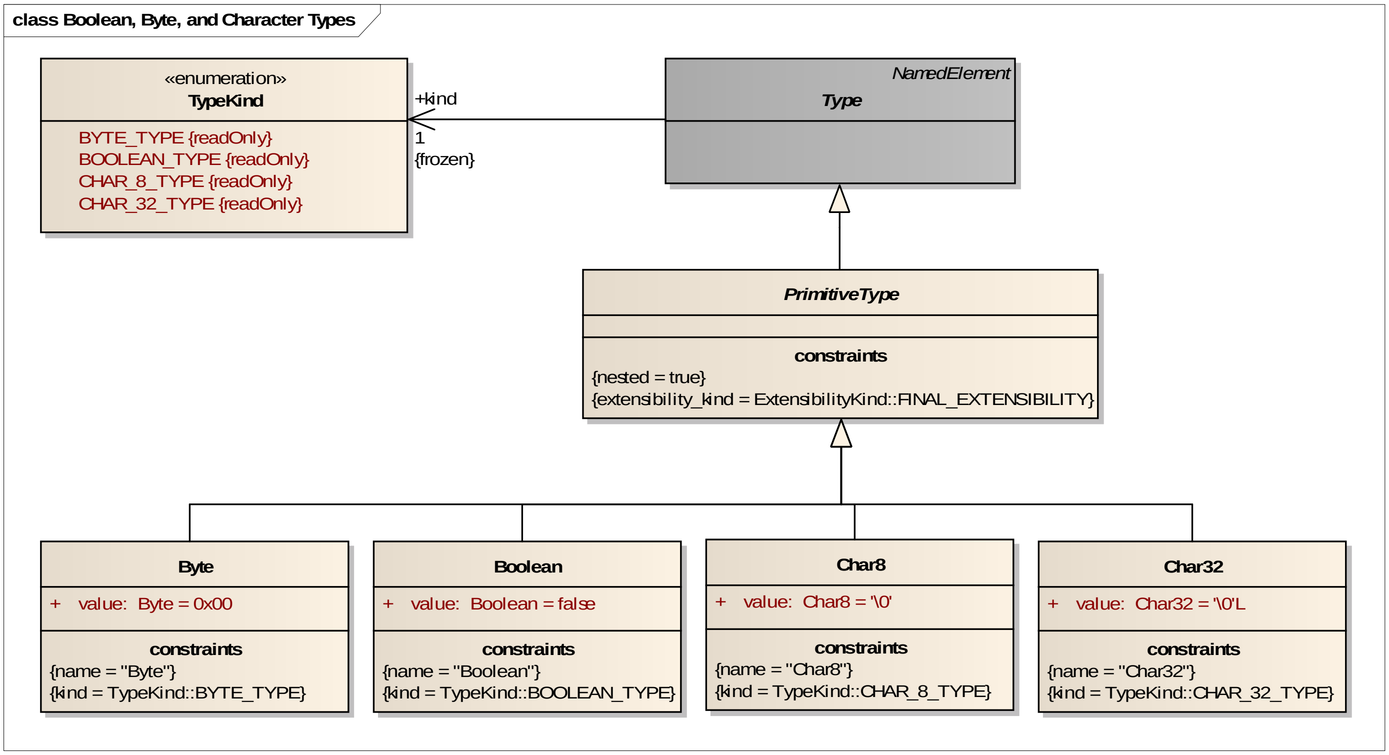

Figure 8 - Primitive Types: Booleans, Bytes, and Characters

> *Primitive Types include the primitive types present in most programming languages, including Boolean, integer, floating point, and character.*

The following table enumerates and describes the available primitive types. Note that value ranges are in this package specified only in terms of upper and lower bounds; data sizes and encodings are the domain of the Type Representation and Data Representation packages.

Table 3 – Primitive Types

Type Kind	*Type Name*	*Description*
`BOOLEAN_TYPE`	`Boolean`	Boolean type. Data of this type can only take two values: true and false.
`BYTE_TYPE`	`Byte`	Single opaque byte. A Byte value has no numeric value.
`INT_16_TYPE`	`Int16`	Signed integer minimally capable of representing values in the range -32738 to +32737.
`UINT_16_TYPE`	`UInt16`	Unsigned integer minimally capable of representing values in the range 0 to +65535.
`INT_32_TYPE`	`Int32`	Signed integer minimally capable of representing values in the range -2147483648 to +2147483647.
`UINT_32_TYPE`	`UInt32`	Unsigned integer minimally capable of representing values in the range 0 to +4294967295.
`INT_64_TYPE`	`Int64`	Signed integer minimally capable of supporting values in the range -9223372036854775808 to +9223372036854775807.
`UINT_64_TYPE`	`UInt64`	Unsigned integer minimally capable of supporting values in the range 0 to +18446744073709551617.

`FLOAT_32_TYPE`	`Float32`	Floating point number minimally capable of supporting the range and precision of an IEEE 754 single-precision floating point value.
`FLOAT_64_TYPE`	`Float64`	Floating point number minimally capable of supporting the range and precision of an IEEE 754 double-precision floating point value.
`FLOAT_128_TYPE`	`Float128`	Floating point number minimally capable of supporting the range and precision of an IEEE 754 quadruple-precision floating point value.
`CHAR_8_TYPE`	`Char8`	Character type minimally capable of supporting the ISO-8859-1 character set.
`CHAR_32_TYPE`	`Char32`	Character type minimally capable of supporting the Universal Character Set (UCS).

The primitive types do not exist within any module; their names are top-level names.

7.2.2.2.1 Character Data

The character types identified above require further definition, provided here.

7.2.2.2.1.1 Design Rationale (Non-Normative)

Because the Unicode character set is a superset of the US-ASCII character set, some readers may question why this specification provides two types for character data: `Char8` and `Char32`. These types are differentiated to facilitate the efficient representation and navigation of character data as well as to more accurately describe the designs of existing systems.

Existing languages for type definition—including C, C++, and IDL—distinguish between regular and wide characters (C/C++ `char` vs. `wchar_t`; IDL `char` vs. `wchar`). While other commonly used typing systems do not make such a distinction—in particular Java and the ECMA Common Type System, of which Microsoft's .Net is an implementation—it is more straightforward to map two platform-independent types to a single platform-specific type than it is to map objects of a single platform-independent type into different platform-specific types based on their values.

7.2.2.2.1.2 Character Sets and Encoding

The ISO-8859-1 character set[1] standard [LATIN], a superset of Latin-1, identifies all possible characters used by `Char8` and `String<Char8>` data. Implementations of these types must therefore provide a minimal level of expressiveness sufficient to represent this character set (eight bits are sufficient).

The Universal Character Set standard [UCS] identifies all possible characters used by the `Char32` and `String<Char32>` data. Implementations of these types must therefore provide a minimal level of expressiveness sufficient to represent this character set (between eight and 32 bits are sufficient, depending on the character).

Although the Type System identifies the characters with which it is concerned, it does not identify the character encoding to be used to represent data defined by that type system. (For example, UTF-8, UTF-16, and UTF-32 are several of the standard encodings for UCS data.) These details are defined by a particular Data Representation.

---

[1] A word about IDL compatibility: IDL defines the graphical characters based on the ISO 8859-1 (Latin-1) character set (note the space in place of the first hyphen) and the non-graphical characters (*e.g.* NUL) based on the ASCII (ISO 646) specification. These two specifications together do not define the meanings of all 256 code points that can be represented by an eight-bit character. The ISO-8859-1 character set (note the extra hyphen) unifies and extends these two earlier standards and defines the previously undefined code points. ISO-8859-1 is the standard default encoding of documents delivered via HTTP with a MIME type beginning with "text/."

7.2.2.3 Constructed Types

The definitions of these types are constructed from—that is, based upon—the definitions of other types. These other types may be either primitive types or other constructed types: type definitions may be recursive to an arbitrary depth. Constructed types are explicitly defined by a user of an implementation of this specification and are assigned a name when they are defined.

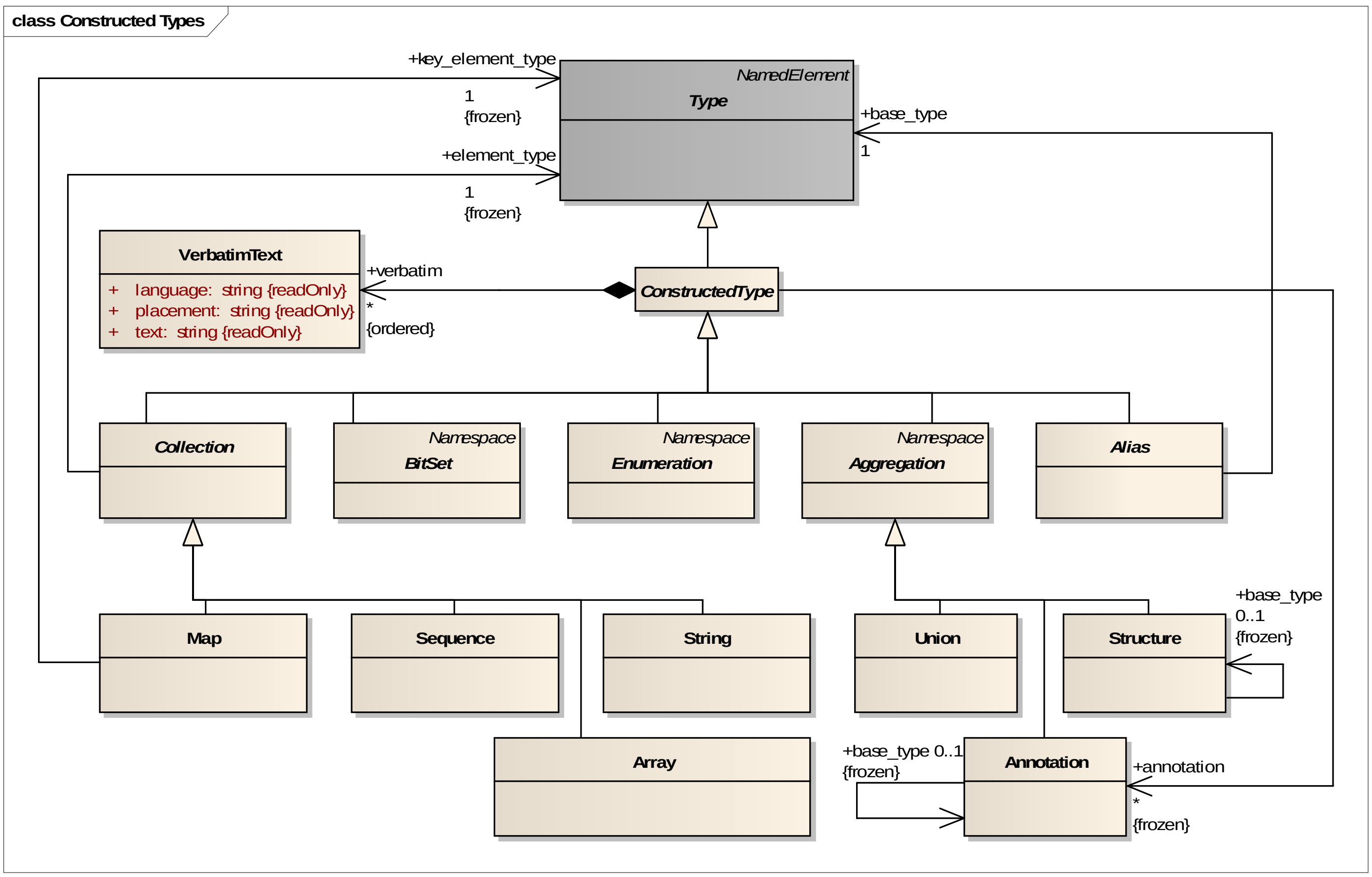

Figure 9 - Constructed Types

There are several kinds of Constructed Types: Collections, Aggregations, Aliases, Bit Sets, and Enumerations. Collections are homogeneous in that all elements of the collection have the same type. Aggregations are heterogeneous; members of the aggregation may have different types. Aliases introduce a new name for another type. Enumerations define a finite set of possible integer values for the data.

7.2.2.3.1 Enumeration Types

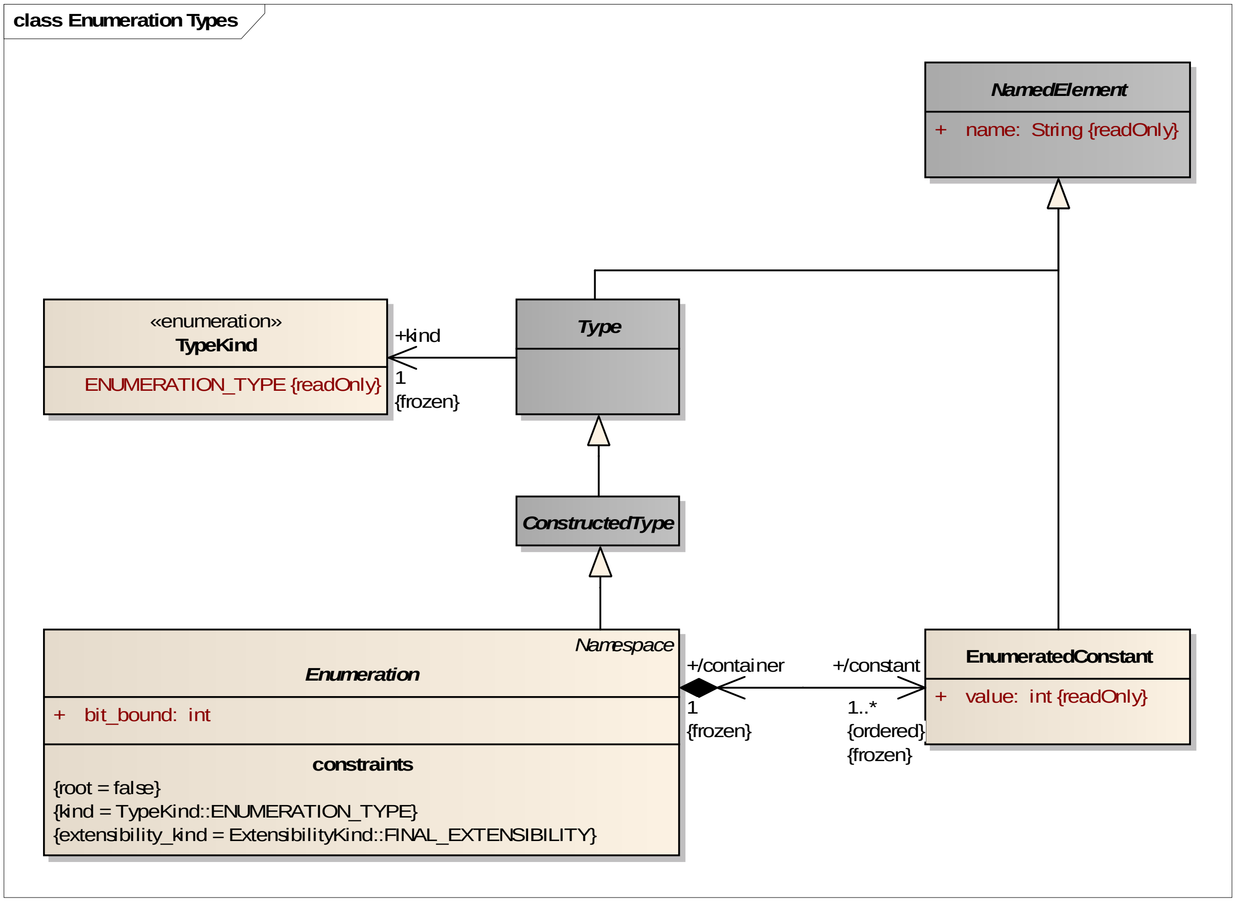

Figure 10 - Enumeration Types

Table 4 - Enumeration types

Type Kind	*Type Name*	*Description*
`ENUMERATION_TYPE`	*Assigned when type is defined*	**Set of constants.** An enumeration type defines a closed set of one or more constant objects of that type. Each object of a given enumeration type has a name and an `Int32` value that are each unique within that type. The order in which the constants of an enumeration type are defined is significant to the definition of that type. For example, some type representations may base the numeric values of the constants on their order of definition.

7.2.2.3.2 BitSet Types

Bit sets, as in the C++ standard library (and not unlike the `EnumSet` class of the Java standard library), represent a collection of Boolean flags, each of which can be inspected and/or set individually.

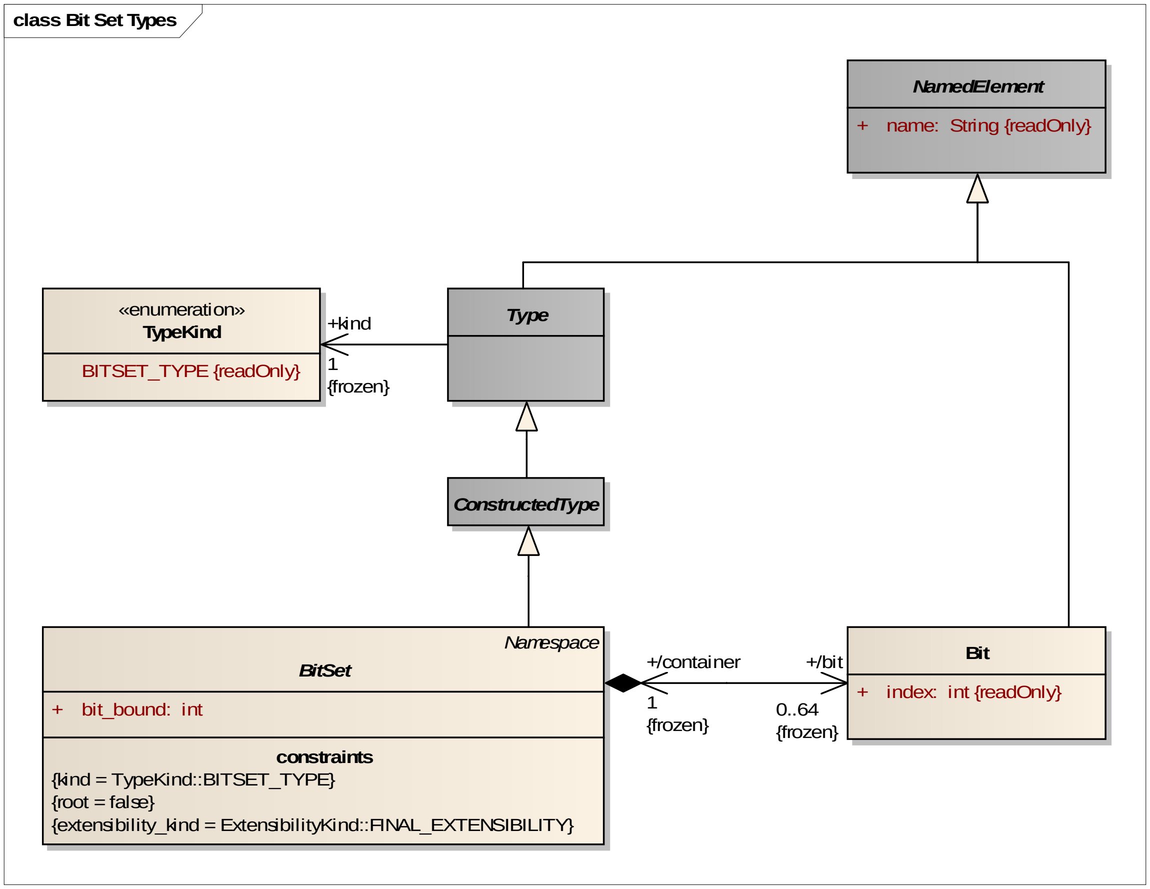

Figure 11 - Bit Set Types

Table 5 - Bit set types

Type Kind	***Type Name***	***Description***
`BITSET_TYPE`	*Assigned when type is defined*	**Ordered set of named Boolean flags.** A bit set defines a bound—the maximum number of bits in the set—and identifies by name certain bits within the set. The bound must be greater than zero and no greater than 64.

A bit set type reserves a number of "bits" (Boolean flags); this is referred to as its bound. (The bound of a bit set is logically similar to the bound of an array, except that the "elements" in a bit set are single bits.) It then identifies some subset of those bits. Each bit in this subset is identified by name and by an index, numbered from 0 to (bound – 1). The bit set need not identify every bit it reserves. Furthermore, the bits it does identify need not be contiguous.

Note that this type exists for the sake of semantic clarity and to enable more efficient data representations. It does not actually constrain such representations to represent each "bit" in the set as a single memory bit or to align the bit set in any particular way.

7.2.2.3.2.1 Design Rationale (Non-Normative)

It is commonly the case that complex data types need to represent a number of Boolean flags. For example, in the DDS specification, status kinds are represented as `StatusKind` bits that are combined into a `StatusMask`. A bit set (also referred to as a bit mask) allows these flags to be represented very compactly—typically as a single bit per flag. Without such a concept in the type system, type designers must choose one of two alternatives:

- Idiomatically define enumerated "kind" bits and a "mask" type. Pack and unpack the former into the latter using bitwise operators. As previously noted, this is the approach taken by the DDS specification in the case of statuses, because it predated this enhanced type model. There are several weaknesses to this approach:
  - It is verbose, both in terms of the type definition and in terms of the code that uses the bit set; this verbosity slows understanding and can lead to programming errors.
  - It is not explicitly tied to the semantics of the data being represented. This weakness can lead to a lack of user understanding and type safety, which in turn can lead to programming errors. It furthermore hampers the development of supporting tooling, which cannot interpret the "bit set" otherwise than as a numeric quantity.
- Represent the flags as individual Boolean values. This approach simplifies programming and provides semantic clarity. However, it is extremely verbose: a structure of Boolean members wastes at least 7/8 of the network bandwidth it uses (assuming no additional alignment and that each flag requires one bit but occupies one byte) and possible up to 31/32 of the memory it uses (on platforms such as Microsoft Windows that conventionally align Boolean values to 32-bit boundaries).

7.2.2.3.3 Alias Types

Alias types introduce an additional name for another type.

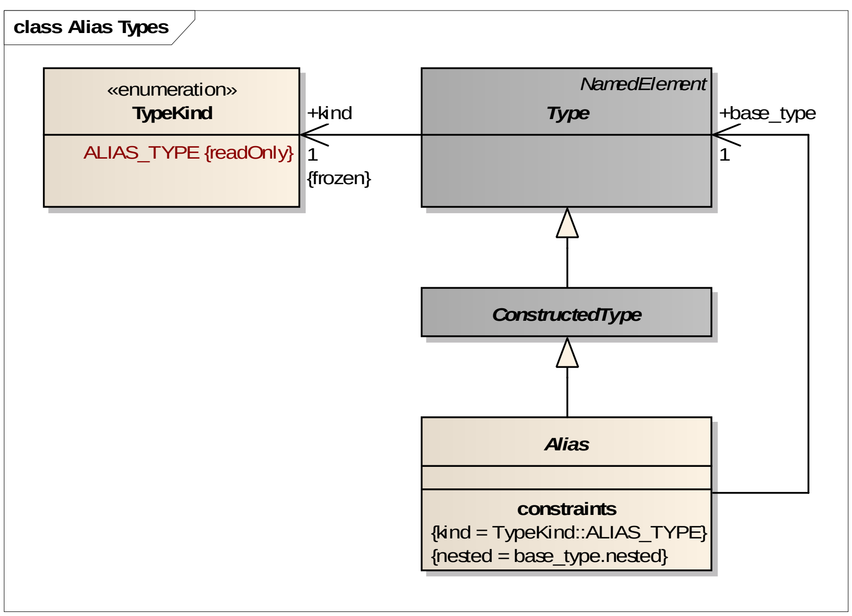

Figure 12 - Alias Types

Table 6 - Alias types

Type Kind	*Type Name*	*Description*
`ALIAS_TYPE`	*Assigned when type is defined*	**Alternative name for another type.** An alias type—also referred to as a *typedef* from its representation in IDL, C, and elsewhere—applies an additional name to an already-existing type. Such an alternative name can be helpful for suggesting particular uses and semantics to human readers, making it easier to repeat complex type names for human writers, and simplifying certain language bindings. As in the C and C++ programming languages, an alias/typedef does not introduce a distinct type. It merely provides an alternative name by which to refer to another type.

7.2.2.3.4 Collection Types

Collections are containers for elements of a homogeneous type. The type of the element might be any other type, primitive or constructed (although some limitations apply; see below) and must be specified when the collection type is defined.

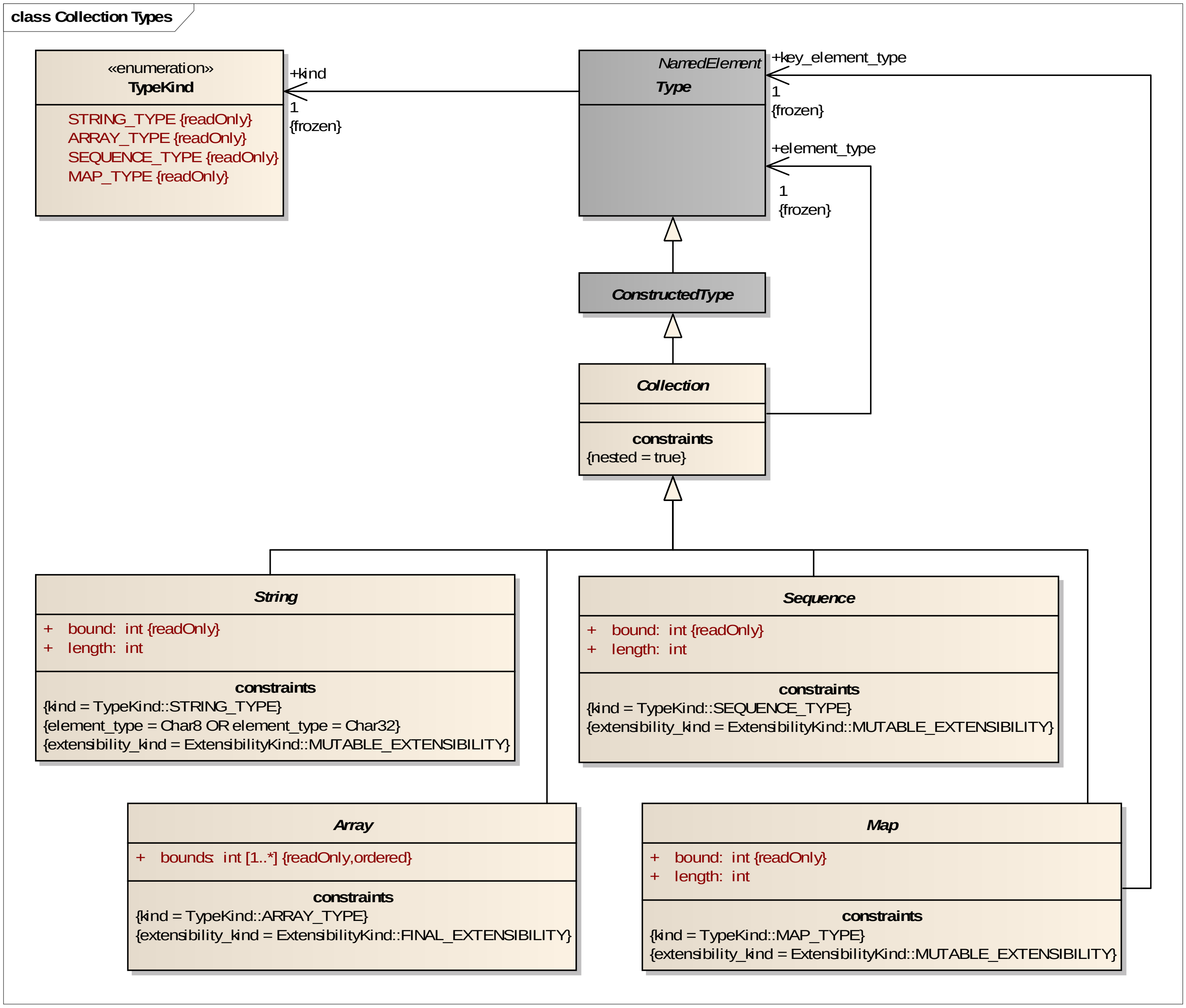

Figure 13 - Collection Types

There are three kinds of Collection Types: ARRAY, SEQUENCE, and MAP. These kinds are described in Table 7.

Table 7 – Collection Types

Type Kind	*Type Name*	*Description*
`ARRAY_TYPE`	*Assigned implicitly*	**Fixed-size multi-dimensional collection.** Arrays are of a fixed size in that all objects of a given array type will have the same number of elements. Elements are addressed by a sequence of indices (one per dimension). Semantically, array types of higher dimensionality are distinct from arrays of arrays of lower dimensionality. (For example, a two-dimensional array is not just an array of one-dimensional arrays.) However, certain type representations may be unable to capture this distinction. (For example, IDL provides no syntax to describe an array of arrays[2], and in Java, all "multi-dimensional" arrays *are* arrays of arrays necessarily.) Such limitations in a given type representation should not be construed as a limitation on the type system itself.
`SEQUENCE_TYPE`	*Assigned implicitly*	**Variable-size single-dimensional collection.** Sequences are variably sized in that objects of a given sequence type can have different numbers of elements (the sequence object's "length"); furthermore, the length of a given sequence object may change between zero and the sequence type's "bound" (see below) over the course of its lifetime. Elements are addressed by a single index.
`STRING_TYPE`	*Assigned implicitly*	**Variable-size single-dimensional collection of characters.** Strings are variably sized in that objects of a given string type can have different numbers of elements (the string object's "length"); furthermore, the length of a given string object may change between zero and the string type's "bound" (see below) over the course of its lifetime. A string is logically very similar to a sequence. However, the element type of a string must be either `Char8` or `Char32` (or an alias to one of these); other element types are undefined. These two collections have been distinguished in order to preserve the fidelity present in common implementation programming languages and platforms.
`MAP_TYPE`	*Assigned implicitly*	**Variable-size associative collection.** Maps are variably sized in that objects of a given map type can have different numbers of elements (the map object's "length"); furthermore, the length of a given map object may change between zero and the map type's "bound" (see below) over the course of its lifetime. "Map value" elements are addressed by a "map key" object, the value of which must be unique within a given map object. The types of both of these are homogeneous within a given map type and must be specified when the map type is defined.

Collection types are defined implicitly as they are used. Their definitions are based on three attributes:

- **Collection kind**: The supported kinds of collections are identified in the table above.
- **Element type**: The concrete type to which all elements conform. (Collection elements that are of a subtype of the element type rather than the element type itself may be truncated when they are serialized into a Data Representation.)

[2] An intermediate alias can help circumvent this limitation; see below for a more formal treatment of aliases.

In the case of a map type, this attribute corresponds to the type of the *value* elements. Map types have an additional attribute, the *key element type*, that indicates the type of the may key objects. Implementers of this specification need only support key elements of signed and unsigned integer types and of narrow and wide string types; the behavior of maps with other key element types is undefined and may not be portable.

- **Bound**: The maximum number of elements the collection may contain (inclusively); it must be greater than zero.

  In the cases of sequences, strings, and maps, the bound parameter may be omitted. If it is omitted, the bound is not specified; such a collection is referred to as "unbounded." (All arrays must be bounded.) In that case, the type may have *no* upper bound—meaning that the collection may contain any number of elements—*or* it may have an *implicit* upper bound imposed by a given type representation (which might, for example, provide only a certain number of bits in which to store the bound) or implementation (which might, for example, impose a smaller default bound than the maximum allowed by the type representation for resource management purposes). Because of this ambiguity, type designers are encouraged to choose an explicit upper bound whenever possible.

  In the cases of sequences, strings, and maps, the bound is a single value. Arrays have independent bounds on each of their dimensions; they can also be said to have an overall bound, which is the product of all of their dimensions' bounds.

For example, a one-dimensional array of 10 integers, a one-dimensional array of 10 short integers, a sequence of at most 10 integers, and a sequence of an unspecified number of integers are all of different types. However, all one-dimensional arrays of 10 integers are of the same type.

Because some standard Type Representations (e.g., IDL) do not allow collection types to be named explicitly, and all Type Representations must be fully capable of expressing any type in the Type System, the Type System does not allow collection type names to be set explicitly. Collection types shall be named automatically based on the three parameters above.

A collection type's implicit name is the concatenation of a label that identifies the type of collection (given below), the bound(s) (for bounded collections, expressed as a decimal integer), the key element type name (for maps), and the element type name, separated by underscores. These names are all in the global namespace.

The collection type labels are:

- "sequence" (for type kind SEQUENCE_TYPE)
- "string" (for type kind STRING_TYPE)
- "map" (for type kind MAP_TYPE)
- "array" (for type kind ARRAY_TYPE)

For example, the following are all valid implicit type names:

- `sequence_10_integer`
- `string_widecharacter`
- `sequence_10_string_15_character`
- `map_20_integer_integer`
- `array_12_8_string_64_character`

7.2.2.3.5 Aggregation Types

Aggregations are containers for elements—"members"—of (potentially) heterogeneous types. Each member is identified by a string name and an integer ID. Each must be unique within a given type. Each member also has a type; this type may be the same as or different than the types of other members of the same aggregation type.

The relative order in which an aggregated type's members are defined is significant, and may be relied upon by certain Data Representations.

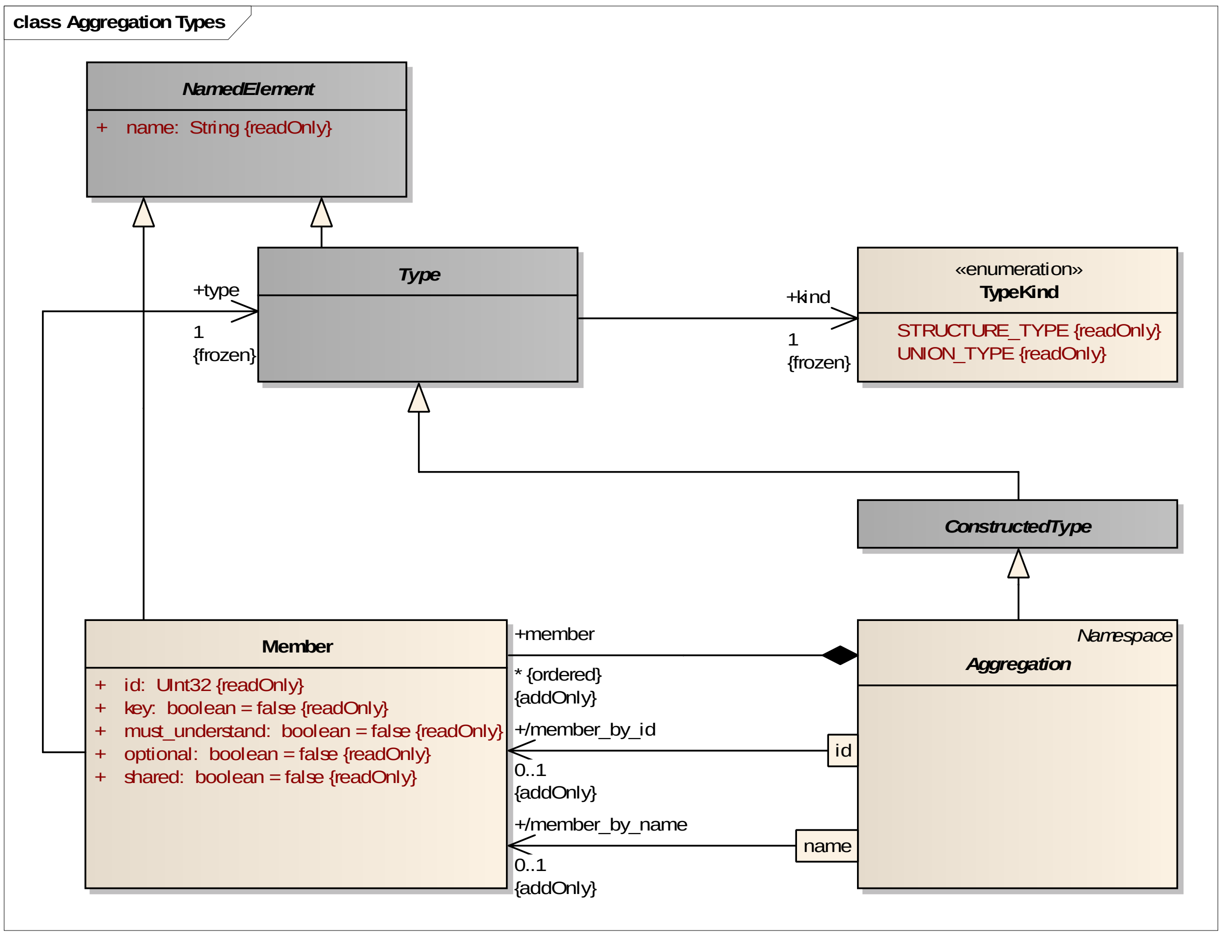

Figure 14 - Aggregation Types

There are three kinds of Aggregation Types: structures, unions, and annotations. These kinds are described in Table 8.

Table 8 - Aggregation Types

Type Kind	*Type Name*	*Description*
`UNION_TYPE`	*Assigned when type is defined*	**Discriminated exclusive aggregation of members.** Unions define a well-known discriminator member and a set of type-specific members.
`STRUCTURE_TYPE`	*Assigned when type is defined*	**Non-exclusive aggregation of members.** A type designer may declare any number of members within a structure. Unlike in a union, there are no implicit members in a structure, and values for multiple members may coexist.

7.2.2.3.5.1 Structure Types

A type designer may declare any number of members within a structure. Unlike in a union, there are no implicit members in a structure, and values for multiple members may coexist.

A structure can optionally extend one other structure, its "`base_type`." In the event that there is a name or ID collision between a structure and its base type, the definition of the member in the former takes precedence.

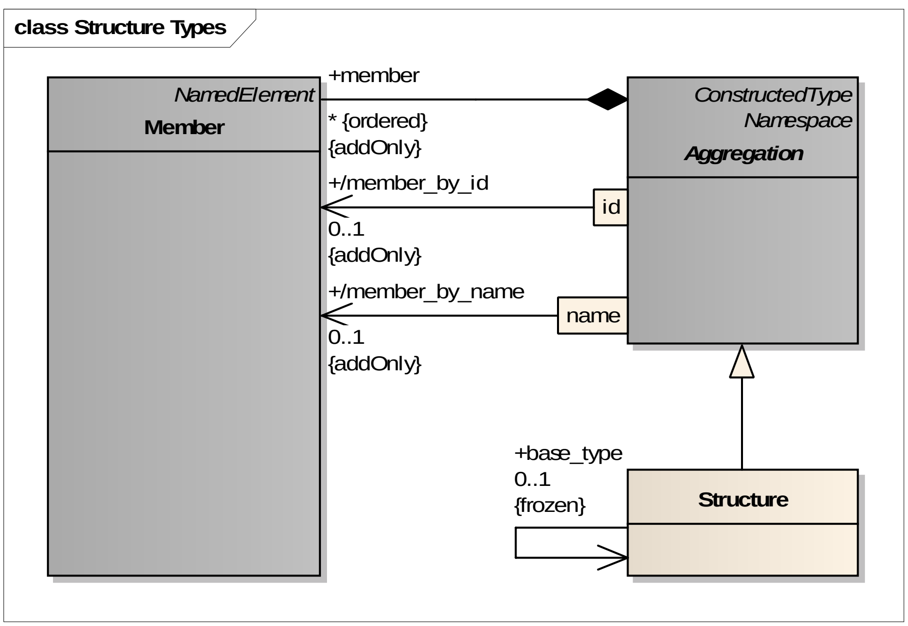

Figure 15 - Structure Types

7.2.2.3.5.2 Union Types

Unions define a well-known discriminator member and a set of type-specific members. The name of the discriminator member is always "discriminator"; that name is reserved for union types and is not permitted for type-specific union members. The discriminator member is always considered to be the first member of a union.

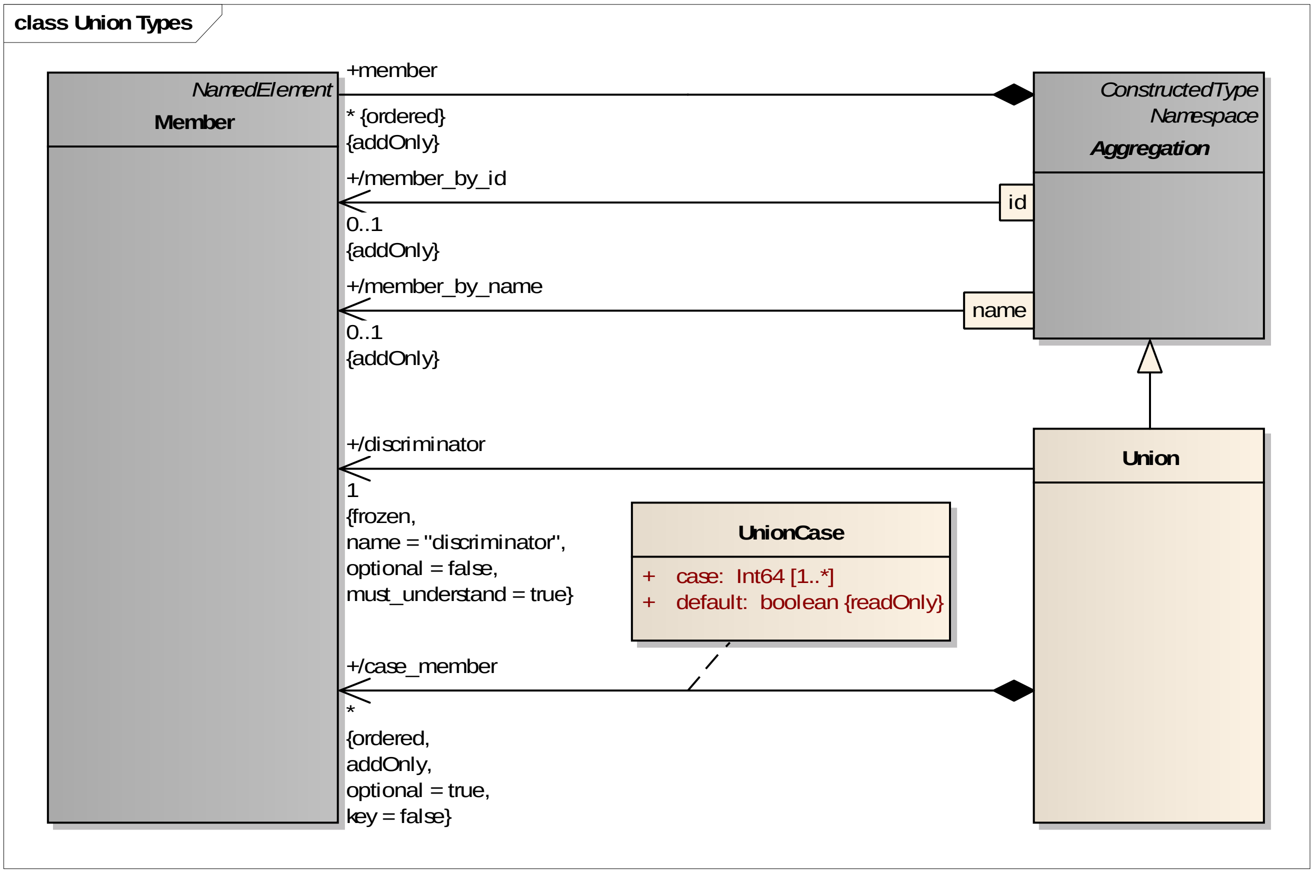

Figure 16 - Union Types

Each type-specific member is associated with one or more values of the discriminator. These values are identified in one of two ways: (*1*) They may be identified explicitly; it is not allowed for multiple members to explicitly identify the same discriminator value. (*2*) At most one member of the union may be identified as the "default" member; any discriminator value that does not explicitly identify another member is considered to identify the default member. These two mechanisms together guarantee that any given discriminator value identifies at most one member of the union. (*Note* that it is not required for every potential discriminator value to be associated with a member.) These mappings from discriminator values to members are defined by a union *type* and do not differ from object to object.

The value of the member associated with the current value of the discriminator is the only member value considered to exist in a given object of a union type at a given moment in time. However, the value of the discriminator field may change over the lifetime of a given object, thereby changing which union member's value is observed. When such a change occurs, the initial value of the newly observed member is undefined by the type system (though it may be defined by a particular language binding). In particular, it is not defined whether, upon switching from a discriminator value *x* to a different value *y* and then immediately back to *x*, the previous value of the *x* member will be preserved.

The discriminator of a union must be of one of the following types:

- `Boolean`
- `Byte`
- `Char8`, `Char32`

- `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`
- Any enumerated type
- Any alias type that resolves, directly or indirectly, to one of the aforementioned types.

7.2.2.3.5.3 Member IDs

As noted above, each member of an aggregated type is uniquely identified within that type by an integer "member ID." Member IDs are unsigned and have a range that can be represented in 28 bits: from zero to 268,435,455 (0x0FFFFFFF). (The full range of a 32-bit unsigned integer is *not* used in order to allow binary Data Representations the freedom to embed a small amount of meta-data into a single 32-bit field if they so desire.)

The upper end of the range, from 268,419,072 (0x0FFFC000) to 268,435,455 (0x0FFFFFFF) inclusive, is reserved for use by the OMG, either by this specification—including future versions of it—or by future related specifications (16,384 values). The largest value in this range—0x0FFFFFFF—shall be used as a sentinel to indicate an invalid member ID. This sentinel is referred to by the name `MEMBER_ID_INVALID`.

The remaining part of the member ID range—from 0 to 268,402,687 (0x0FFFBFFF)—is available for use by application-defined types compliant with this specification.

7.2.2.3.5.4 Members That Must Be Understood by Consumers

A consumer of data may not have the same definition for a type as did the producer of that data. Such a situation may come about as a result of the independent, decoupled definition of the respective types or as a result of a single type's evolution over time. A consumer, upon observing a member value it does not understand, must be able to determine whether it is acceptable to ignore the member and continue processing other members, or whether the entire data sample must be discarded.

Each member of an aggregated type has a Boolean attribute "must understand" that satisfies this requirement. If the attribute is true, a data consumer, upon identifying a member it does not recognize, must discard the entire data sample to which the member belongs. If the attribute is false, the consumer is permitted to process the sample, omitting the value of the unrecognized member.

In a structure type, each member may have the "must understand" attribute set to true or false independently.

In a union type, the discriminator member shall always have the "must understand" attribute set to true.

The ability of a consumer to detect the presence of an unrecognized member depends on the Data Representation. Each representation shall therefore define the means by which such detection occurs.

7.2.2.3.5.5 Optional Members

Each member of an aggregated type has a Boolean attribute that indicates whether it is *optional*. Every object of a given type shall be considered to contain a value for every non-optional member defined by that type. In the event that no explicit value for such a member is ever provided in a Data Representation of that object, that member is considered to nevertheless have the default "zero" value defined in the following table:

Table 9 - Default values for non-optional members

Type Kind	***Default Value***
`BYTE`	0x00
`BOOLEAN`	False
`INT_16_TYPE`,	0

UINT_16_TYPE, INT_32_TYPE, UINT_32_TYPE, INT_64_TYPE, UINT_64_TYPE, FLOAT_32_TYPE, FLOAT_64_TYPE, FLOAT_128_TYPE	
CHAR_8_TYPE, CHAR_32_TYPE	'\0'
STRING_TYPE	""
ARRAY_TYPE	*An array of the same dimensions and same element type whose elements take the default value for their corresponding type.*
ALIAS_TYPE	*The default type of the alias's base type.*
BITSET_TYPE	*All bits, identified or merely reserved, set to zero.*
SEQUENCE_TYPE	*A zero-length sequence of the same element type.*
MAP_TYPE	*An empty map of the same element type.*
ENUM_TYPE	*The first value in the enumeration.*
UNION_TYPE	*A union with the discriminator set to select the default element, if one is defined, or otherwise to the lowest value associated with any member. The value of that member set to the default value for its corresponding type.*
STRUCTURE_TYPE	*A structure without any of the optional members and with other members set to their default values based on their corresponding types.*

An object may omit a value for any optional member(s) defined by its type. Omitting a value is semantically similar to assigning a null value to a pointer in a programming language: it indicates that no value exists or is relevant. Implementations shall *not* provide a default value in such a case.

The discriminator member of a union shall never be optional. The other members of a union shall always be optional. The designer of a structure can choose which members are optional on a member-by-member basis.

The value of a member's "optional" attribute is unrelated to the value of its "must understand" attribute. For example, it is legal to define a type in which a non-optional member can be safely skipped or one in which an optional member, if present and not understood, must lead to the entire sample being discarded.

7.2.2.3.5.6 Key Members

A given member of an aggregated type may be designated as part of that type's *key*. The type's key will become the key of any DDS Topic that is constructed using the aforementioned aggregated type as the Topic's type. If a given type has no members designated as key members, then the type—and any DDS Topic that is constructed using it as its type it—has no key.

Key members shall never be optional, and they shall always have their "must understand" attribute set to true.

Which members may together constitute a type's key depends on that type's kind. In a structure type, the key designation can be applied to any member and to any number of members. In a union type, only the discriminator is permitted to be a key member.

In the event that the type *K* of a key member of a given type *T* itself defines key members, only the key of *K*, and not any other of its members, shall be considered part of the key of *T*. This relationship is recursive: the key members of *K* may themselves have nested key members. For example, suppose the key of a medical record is a structure describing the individual whose record it is. Suppose also that the nested structure (the one describing the individual) has a key member that is the social security number of that individual. The key of the medical record is therefore the social security number of the person whose medical record it is.

7.2.2.3.5.7 Shareable Data

In some cases, it is necessary and/or desirable to provide information to a language binding that a certain member's data should be stored, not inline within its containing type, but external to it (e.g., using a pointer).

- For example, the data may be very large, such that it is impractical to copy it into a sample object before sending it on the network. Instead, it is desirable to manage the storage outside of the middleware and assign a reference in the sample object to this external storage.
- For example, the type of the member may be the type of a containing type (directly or indirectly). This will be the case when defining linked lists or any of a number of more complex data structures.

Type Representations shall therefore allow the following type relationships in the case of shareable members, which would typically cause errors in the case of non-shareable members:

- A shareable member of an aggregated type shall be permitted to refer to a type whose definition is incomplete (*i.e.* is identified only by a forward declaration) at the time of the member's declaration.
- A shareable member of an aggregated type shall be permitted to refer to the member's containing type.

Each member of an aggregated type—with the exception of the discriminator of a union type—may be optionally marked as *shareable*.

Note that this attribute does *not* provide a means for modeling object graphs.

7.2.2.3.6 Annotation Types

Annotation types are aggregation types, strictly speaking. However, they are different from structures and unions in that objects of these types are encountered at compile time, not at runtime.

Table 10 - Annotation types

Type Kind	***Type Name***	***Description***
`ANNOTATION_TYPE`	*Assigned when type is defined*	**Non-exclusive aggregation of members instantiated at compile time.** An annotation describes a piece of metadata attached to a type or type member. An annotation type defines the structure of the metadata. That type is "instantiated," and its members given values, within the representation of another type when the annotation is applied to an element of that other type.

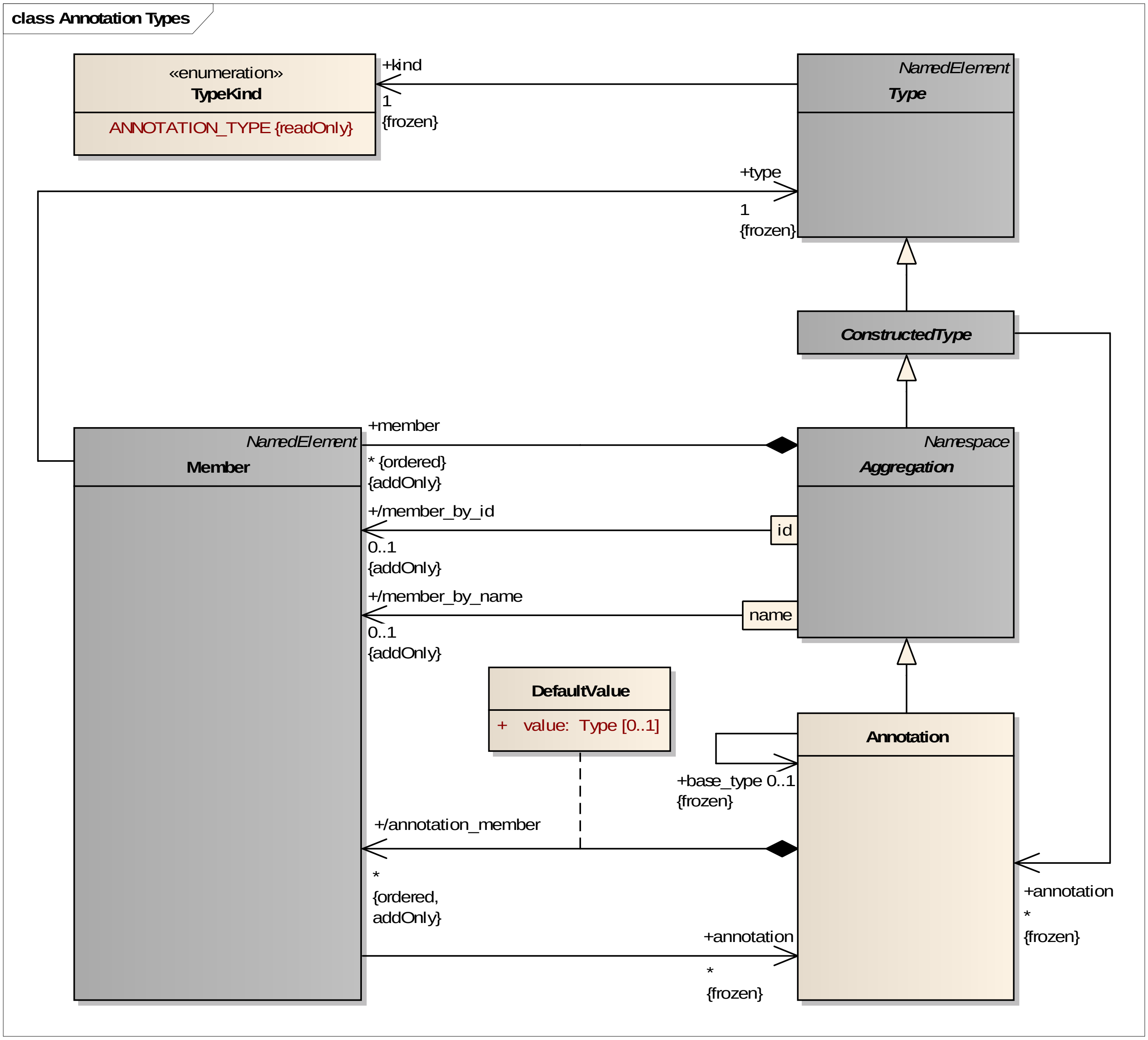

Figure 17 - Annotation Types

Unlike members of other aggregated types, members of annotations can have custom default values. Because the compiler of a Type Representation must be able to efficiently interpret an annotation instantiation, and because member default values must be easily expressed as object literals in a variety of Type Representations, the members of annotation types are restricted to certain types.

These are:

- Any Primitive type
- Any `String` type of `Char8` or `Char32` elements
- Any enumerated type

Like structure types, annotation types support single inheritance. Note that structures can subtype other structures, and annotations can subtype other annotations, but structures cannot subtype annotations or visa versa.

Furthermore, because annotations are interpreted at compile time, they cannot be used to type objects that will exist at runtime, such as members of other aggregated types.

7.2.2.3.7 Verbatim Text

System developers frequently require the ability to inject their own output into that produced by a Type Representation compiler. Such output typically depends on the target programming language, not on the Type Representation. Furthermore, it is desirable to be able to preserve information about such output across translations of the Type Representation. Therefore, it is appropriate to manage user-specified content within the Type System for use by all Type Representations and therefore by Type Representation compilers. The `VerbatimText` class serves this purpose; each constructed type may refer to one or more instances of this class.

A `VerbatimText` object defines three properties; each is a string:

- `language`: The target programming language for which the output text applies.
- `placement`: The location within the generated output at which the output text should be inserted.
- `text`: The literal output text to be copied into the output by the Type Representation compiler.

7.2.2.3.7.1 Property: Language

When a Type Representation compiler generates code for the programming language named (case-insensitively) by this property, it shall copy the string contained in the `text` property into its output.

- The string “c” shall indicate the C programming language [C-LANG].
- The string “c++” shall indicate the C++ programming language [C++-LANG].
- The string “java” shall indicate the Java programming language [JAVA-LANG].
- The string “*” (an asterisk) shall indicate that `text` applies to all programming languages.

7.2.2.3.7.2 Property: Placement

This string identifies where, relative to its other output, the Type Representation compiler shall copy the `text` string. It shall be interpreted in a case-insensitive manner. All Type Representation compilers shall recognize the following placement strings; individual compiler implementations may recognize others in addition.

- `begin-declaration-file`: The `text` string shall be copied at the beginning of the file containing the declaration of the associated type before any type declarations.

  For example, a system implementer may use such a `VerbatimText` instance to inject import statements into Java output that are required by literal code inserted by other `VerbatimText` instances.

- `before-declaration`: The `text` string shall be copied immediately before the declaration of the associated type.

  For example, a system implementer may use such a `VerbatimText` instance to inject documentation comments into the output.

- `begin-declaration`: The `text` string shall be copied into the body of the declaration of the associated type before any members or constants.

  For example, a system implementer may use such a `VerbatimText` instance to inject additional declarations or implementation into the output.

- `end-declaration`: The `text` string shall be copied into the body of the declaration of the associated type after all members or constants.
- `after-declaration`: The `text` string shall be copied immediately after the declaration of the associated type.
- `end-declaration-file`: The `text` string shall be copied at the end of the file containing the declaration of the associated type after all type declarations.

7.2.2.3.7.3 Property: Text

The Type Representation compiler shall copy the string contained in this property into its output as described above.

7.2.2.4 Nested Types

Not every type in a user's application will be used to type DDS Topics; some types appear only as the types of members within other types. It is desirable to distinguish these two cases for the same of efficiency; for example, an IDL compiler need not generate typed `DataWriter`, `DataReader`, and `TypeSupport` classes for types that are not intended to type topics. Types that are not intended to describe topic data are referred to as *nested* types.

7.2.3 Type Extensibility and Mutability

In some cases, it is desirable for types to evolve without breaking interoperability with deployed components already using those types. For example:

- A new set of applications to be integrated into an existing system may want to introduce additional fields into a structure. These new fields can be safely ignored by already deployed applications, but applications that do understand the new fields can benefit from their presence.
- A new set of applications to be integrated into an existing system may want to increase the maximum size of some sequence or string in a Type. Existing applications can receive data samples from these new applications as long as the actual number of elements (or length of the strings) in the received data sample does not exceed what the receiving applications expects. If a received data sample exceeds the limits expected by the receiving application, then the sample can be safely ignored (filtered out) by the receiver.

In order to support use cases such as these, the type system introduces the concept of *extensible* and *mutable* types.

- A type may be *final*, indicating that the range of its possible data values is strictly defined. In particular, it is not possible to add elements to members of collection or aggregated types while maintaining type assignability.
- A type may be *extensible*, indicating that two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable.

- A type may be *mutable*, indicating that two types may differ from one another in the additional, removal, and/or transposition of elements/members while remaining assignable.

This attribute may be used by the Data Representations to modify the encoding of the type in order to support its extensibility.

The meaning of these extensibility kinds is formally defined with respect to type compatibility in section 7.2.4, "Type Compatibility: "is-assignable-from" relationship." It is summarized more generally in Table 11.

Table 11 - Meaning of marking types as extensible

Type Kind	***Meaning of marking type as extensible***
Aggregation Types: `STRUCTURE_TYPE`, `UNION_TYPE`, `ANNOTATION_TYPE`	Aggregation types may be final, extensible, or mutable on a type-by-type basis. However, the extensibility kind of a structure type with a base type must match that of the base type. It shall not be permitted for a subtype to change the extensibility kind of its base type. Any members marked as keys must be present in all variants of the type.
Collection Types: `ARRAY_TYPE`, `SEQUENCE_TYPE`, `STRING_TYPE`, `MAP_TYPE`	String, sequence, and map types are always mutable. Array types are always final. Variations of a mutable collection type may change the maximum number of elements in the collection.
`ENUMERATION_TYPE`	Enumeration types may be final, extensible, or mutable on a type-by-type basis.
`BITSET_TYPE`	Bit set types are always final.
`ALIAS_TYPE`	Since aliases are semantically equivalent to their base types, the extensibility kind of an alias is always equal to that of its base type.
Primitive types	Primitive types are always final.

7.2.4 Type Compatibility: "is-assignable-from" relationship

In order to maintain the loose coupling between data producers and consumers, especially as systems change over time, it is desirable that the two be permitted to use slightly different versions of a type, and that the infrastructure perform any necessary translation. To support type evolution and inheritance the type system defines the "is-assignable-from" directed binary relationship between every pair of types in the Type System.

Given two types T1 and T2, we will write:

```
T1  is-assignable-from  T2
```

…if and only T1 is related to T2 by this relationship. The rules to determine whether two types thus related are given in the following tables.

Intuitively, if T1 *is-assignable-from* T2, it means that in general it is possible, in a structural way, to set the contents of an object of type T1 to the contents of an object of T2 (or perhaps a subset of those contents, as defined below) without leading to incorrect interpretations of that information.

This does not mean that *all* objects of T2 can be assigned to T1 objects (for example, a collection may have too many elements) but that the difference between T2 and T1 is such that (*a*) a meaningful subset of T2 objects will be assignable without misinterpretation and that (*b*) the remaining objects of T2–which are referred to as "unassignable to T1"—can be detected as such so that misinterpretations can be prevented. For the same of run-time efficiency, these per-object assignability limitations are designed such that their enforcement does not require any inspection of a data producer's type definition. Per-object enforcement can potentially be avoided altogether—depending on the implementation—by declaring a type to be *final*[3], forcing producer and consumer types to match exactly; see Section 7.2.3.

In the case T1 *is-assignable-from* T2 but an object of type T2 is encountered that cannot be represented using type T1, the object shall be discarded (filtered out) to avoid misinterpretation.

For example:

T1	***T2***	***T1 is-assignable-from T2?***
Sequence of 10 integers	Sequence of 5 integers	**Yes**. Any object of type T2 can have at most 5 elements; therefore, it can be represented using a sequence of bound 10.
Sequence 10 integers	Sequence of 20 integers	**Yes**. While some objects of type T2 will cannot be represented as T1 (*i.e.* any object with 11 or more elements), there is a sufficiently large subset that are that it is sensible to allow a system to be designed in this manner.

Figure 18 - Type assignability example

Any type is assignable from itself.

If types T1 and T2 are both mutable and T1 *is-assignable-from* T2, then T1 is said to be “strongly” assignable from T2. Strong assignability is an important property in many cases, because it allows consumers of a Data Representation to reliably delimit objects within the Representation and thereby avoid misinterpreting the data.

7.2.4.1 Alias Types

T1 Type Kind	***T2 Type Kinds for which T1 is-assignable-from T2 Is True***	***Behavior***
`ALIAS_TYPE`	*Any type kind* if and only if T1.base_type *is-assignable-from* T2	Transform according to the rules for T1.base_type *is-assignable-from* T2
Any type kind	`ALIAS_TYPE` if and only if T1 *is-assignable-from* T2.base_type	Transform according to the rules for T1 *is-assignable-from* T2.base_type

Figure 19 - Definition of the *is-assignable-from* relationship for alias types

For the purpose of evaluating the *is-assignable-from* relationship, aliases are considered to be fully resolved to their ultimate base types. For this reason, alias types are not discussed explicitly in the subsequent sections. Instead, if T is an alias type, then it shall be treated as if T == T.base_type.

7.2.4.2 Primitive Types

The following table defines the *is-assignable-from* relationship for Primitive Types. These conversions are designed to preserve the data during translation. Furthermore, in order to preserve high performance, they are designed to enable the preservation of *data representation*, such that a `DataReader` is not required to parse incoming samples differently based on the `DataWriter` from which they originate. (For example, although a short integer could be promoted to a long integer without destroying infor-

---

[3] DDS-based systems have an additional tool to enforce stricter static type consistency enforcement: the `TypeConsistencyEnforcementQosPolicy`. See Section 7.6.1.3.

mation, a binary Data Representation is likely to use different amounts of space to represent these two data types. If, upon receiving each sample from the network, a `DataReader` does not consult the type definition of the `DataWriter` that sent that sample, it would not know how many bytes to read. The runtime expense of this kind of type introspection on the critical path is undesirable.)

Table 12 - Definition of the *is-assignable-from* relationship for primitive types

T1 Type Kind	***T2 Type Kinds for which T1 is-assignable-from T2 Is True***	***Behavior***
Any Primitive Type	The same Primitive Type	*Identity*

7.2.4.3 Collection Types

The *is-assignable-from* relationship for collection types is based in part on the same relationship as applied to their element types.

Table 13 - Definition of the *is-assignable-from* relationship for collection types

T1 Type Kind	***T2 Type Kinds for which T1 is-assignable-from T2 Is True***	***Behavior***
`STRING_TYPE`	`STRING_TYPE` if and only if T1.element_type *is-assignable-from* T2.element_type	Assign each character. T1.length is set to T2.length. T2 strings of length greater than T1.bound are unassignable to T1.
`ARRAY_TYPE`	`ARRAY_TYPE` if and only if[4]: • T1.bounds[] == T2.bounds[] • T1.element_type is strongly assignable from T2.element_type	Assign each element. If an element of T2 is unassignable, the whole array is unassignable.
`SEQUENCE_TYPE`	`SEQUENCE_TYPE` if and only if T1.element_type is strongly assignable from T2.element_type	Assign each element. T1.length is set to T2.length. T2 sequences of length greater than T1.bound are unassignable. If an element of T2 is unassignable, the whole sequence is unassignable.
`MAP_TYPE`	`MAP_TYPE` if and only if: • T1.key_element_type is strongly assignable from T2.key_element_type	The result shall be as if the T1 map were cleared of all elements and subsequently all T2 map entries were added to it. The entries are not logically ordered.

[4] Design rationale: This specification allows sequence, map, and string bounds to change but not array bounds. This is because of the desire to avoid requiring the consultation of per-`DataWriter` type definitions during sample deserialization. Without such consultation, a reader of a compact data representation (such as CDR) will have no way of knowing what the intended bound is. Such is not the case for other collection types, which in CDR are prefixed with their length.

	• T1.element_type is strongly assignable from T2.element_type	T2 maps of length greater than T1.bound are unassignable. If a key or value element of T2 is unassignable, the whole map is unassignable.

7.2.4.3.1 Example: Strings

According to the above rules, any string type of narrow characters is assignable to any other string type of narrow characters. Any string type of wide characters is assignable to any other string type of wide characters. However, string types of narrow characters are not assignable from string types of wide characters, or vice versa, because of the possibility of data misinterpretation. For example, suppose a string of wide characters is encoded using the CDR Data Representation. If a consumer of strings of narrow characters were to attempt to consume that string, it might read consider the first byte of the first character to be a character onto itself, the second byte of the first character to be a second character, and so on. The result would be a string of narrow characters having “junk” contents.

Furthermore, any T2 string *object* containing more characters than the bound of the T1 string type is unassignable in order to prevent data misinterpretations resulting from truncations. For example, consider two versions of a shopping list application. The list of purchases is represented by a sequence of strings. Version 2.0 of the application increased the bounds of these strings. Supposing that the list items “cat food” and “catsup” were too long to be understood by a version 1.0 consumer, it would be better to come home from the store without either item than to come home with two cats instead.

7.2.4.4 BitSet and Enumeration Types

Conversions of alias, bit set, and enumeration types are designed to preserve the data during translation.

Table 14 - Definition of the *is-assignable-from* relationship for alias, bit set, and enumeration types

T1 Type Kind	*T2 Type Kinds for which T1 is-assignable-from T2 Is True*	*Behavior*
`BITSET_TYPE`	`BITSET_TYPE` if and only if T1.bound == T2.bound	Preserve bit values by index for all bits identified in both T1 and T2.
	`UINT_32_TYPE` if and only if T1.bound is between 17 and 32, inclusive.	
	`UINT_16_TYPE` if and only if T1.bound is between 9 and 16, inclusive.	
	`UINT_64_TYPE` if and only if T1.bound is between 33 and 64, inclusive.	

	BYTE if and only if T1.bound is between 1 and 8, inclusive.	
ENUMERATION_TYPE	ENUMERATION_TYPE if an only if: • Any constants that have the same name in T1 and T2 also have the same value, and any constants that have the same value in T1 and T2 also have the same name[5]. • T1.extensibility == T2.extensibility AND if T1 is extensible, for each constant index 'i' in T1 the constant in T1 at that index c1[i] and the constant in T2 at that index c2[i], if c2[i] exists, have the same name. AND if T1 is final, the following are also true: • The number of constants in T1 is equal to the number of constants in T2. • For each constant index 'i' in T1 the constant in T1 at that index c1[i] and the constant in T2 at that index c2[i] have the same name.	Choose the corresponding T1 constant if it exists. If the name or value of the T2 object does not exist in T1, the object is unassignable.

7.2.4.5 Aggregation Types

For aggregation types, *is-assignable-from* is based on the same relationship between the types' members. The correspondence between members in the two types is established based on their respective member IDs and on their respective member names.

Table 15 - Definition of the *is-assignable-from* relationship for aggregated types

T1 Type Kind	***T2 Type Kinds for which T1 is-assignable-from T2 Is True***	***Behavior***
UNION_TYPE	UNION_TYPE if and only if it is possible, given any possible discriminator value in T2, to identify the appropriate member in T1 and to transform the T2 member into the T1 member. Specifically: • T1.discriminator.id == T2.discriminator.id and T1.discriminator.type *is-assignable-from* T2.discriminator.type • For every discriminator value in T2 that selects some member m1 of T1, the type of the member m2 of T2 selected by that discriminator value must satisfy m1.type *is-assignable-from* m2.type.	The discriminator of the T1 object takes the value of the discriminator of the T2 object. If the discriminator value selects a member m1 in T1 (where m1 may be the default member) then m1 takes the value of the selected member in T2. If the discriminator value does not select a member in T1 then the T2 object is unassignable to T1.

[5] **Design rationale (non-normative)**: Certain Data Representations may preserve only the value (e.g., CDR) or only the name (e.g., XML). To preserve representation independence, the Type System requires both to remain stable.

[6] **Design rationale (non-normative)**: In practice, default cases severely restrict the evolution of union types. However, more flexible rules would not be computationally practical. Therefore, systems that find these rules overly restrictive should avoid the use of default cases.

	• For every discriminator value that appears in a non-default label in T2 that selects a member m1 of T1, the member m2 of T2 selected by that discriminator value satisfies m1.type *is-assignable-from* m2.type. • If T2 has a default label, it must be true that the type of every member of T1 is assignable from the type of the default member of T2[6].	
`STRUCTURE_TYPE`	`STRUCTURE_TYPE` if and only if: • T1 and T2 have the same number of members in their respective keys. • For each member "m1" that forms part of the key of T1 (directly or indirectly), there is a corresponding member "m2" that forms part of the key of T2 (directly or indirectly) with the same member id (m1.id == m2.id) where m1.type *is-assignable-from* m2.type. (The previous two rules assure that the key of T2 can be transformed faithfully into the key of T1 without aliasing or loss of information.) • Any members in T1 and T2 that have the same a name also have the same ID and any members with the same ID also have the same name. • For each member "m1" in T1, if there is a member m2 in T2 with the same member ID then m1.type *is-assignable-from* m2.type. • For each member "m2" in T2 for which both `optional` is false and `must_understand` is true there is a corresponding member "m1" in T1 with the same member ID. • T1.extensibility == T2.extensibility AND if T1 is extensible, for each member index 'i' in T1 the member in T1 at that index m1[i] and the member in T2 at that index m2[i], if m2[i] exists, have the same member ID and m1[i].type is strongly assignable from m2[i].type. AND if T1 is final, the following are also true: • The number of members in T1 is equal to the number of members in T2. • For each member index 'i' in T1 the member in T1 at that index m1[i] and the member in T2 at that index m2[i] have the same member ID and m1[i].type is strongly assignable from m2[i].type.	Each member "m1" of the T1 object takes the value of the T2 member with the same ID or name, if such a member exists. Each non-optional member in a T1 object that is not present in the T2 object takes the default value. Each optional member in a T1 object that is not present in the T2 object takes no value. If a "must understand" member in the T2 object is present, then T1 must have a member with the same member ID. Otherwise the object is unassignable to T1.

	For the purposes of the above conditions, members belonging to base types of T1 or T2 shall be considered “expanded” inside T1 or T2 respectively, as if they had been directly defined as part of the sub-type.	

7.2.4.5.1 Example: Type Truncation

Consider the following type for representing two-dimensional Cartesian coordinates:

```
struct Coordinate2D {
   long x;
   long y;
};
```

(This example uses the IDL Type Representation. However, the same principles apply to any other type representation.)

Now suppose that another subsystem is to be integrated. That subsystem is capable of representing three-dimensional coordinates:

```
struct Coordinate3D {
   long x;
   long y;
   long z;
};
```

(The type `Coordinate3D` may represent a new version of the `Coordinate2D` type, or the two coordinate types may have been developed concurrently and independently. In either case, the same rules apply.)

`Coordinate2D` is assignable from `Coordinate3D`, because that subset of `Coordinate3D` that is meaningful to consumers of `Coordinate2D` can be extracted unambiguously. In this case, consumers of `Coordinate2D` will observe the two-dimensional projection of a `Coordinate3D`: they will observe the `x` and `y` members and ignore the `z` member.

7.2.4.5.2 Example: Type Inheritance

Type inheritance is a special case of type truncation, which allows objects of subtypes to be substituted in place of objects of supertypes in the conventional object-oriented fashion.

Consider the following type hierarchy:

```
<struct name="Vehicle">
   <member name="km_per_hour" type="int32"/>
</struct>

<struct name="LandVehicle" baseType="Vehicle">
   <member name="num_wheels" type="int32"/>
</struct>
```

(This example uses the XML Type Representation. However, the same principles apply to any other type representation.)

`LandVehicle` is assignable from `Vehicle`. Any consumer of the latter that receives an instance of the former will observe the value of the member `km_per_hour` and ignore the member `num_wheels`.

7.2.4.5.3 Example: Type Refactoring

As systems evolve, it is sometimes desirable to refactor data from place in a type hierarchy to another place. For example, consider the following representation of a giraffe:

```
struct Animal {
   long body_length;
   long num_legs;
};

struct Giraffe : Animal {
   long neck_length;
};
```

(This example uses the IDL Type Representation. However, the same principles apply to any other type representation.)

Now suppose that a later version of the system needs to model snakes in addition to giraffes. Snakes are also animals, but they don't have legs. We could just say that they have zero legs, but then should we add `num_scales` to `Animal` and set that to zero for giraffes? It would be better to refactor the model to capture the fact that legs are irrelevant to snakes:

```
struct Animal {
   long body_length;
};

struct Mammal : Animal {
   long num_legs;
};

struct Giraffe : Mammal {
   long neck_length;
};

struct Snake : Animal {
   long num_scales;
};
```

Because the is-assignable-from relationship is evaluated as if all member definitions were flattened into the types under evaluation, the both versions of the `Giraffe` type are assignable to one another. Producers of one can communicate seamlessly with consumers of the other and correctly observe values for all fields.

7.3 Type Representation

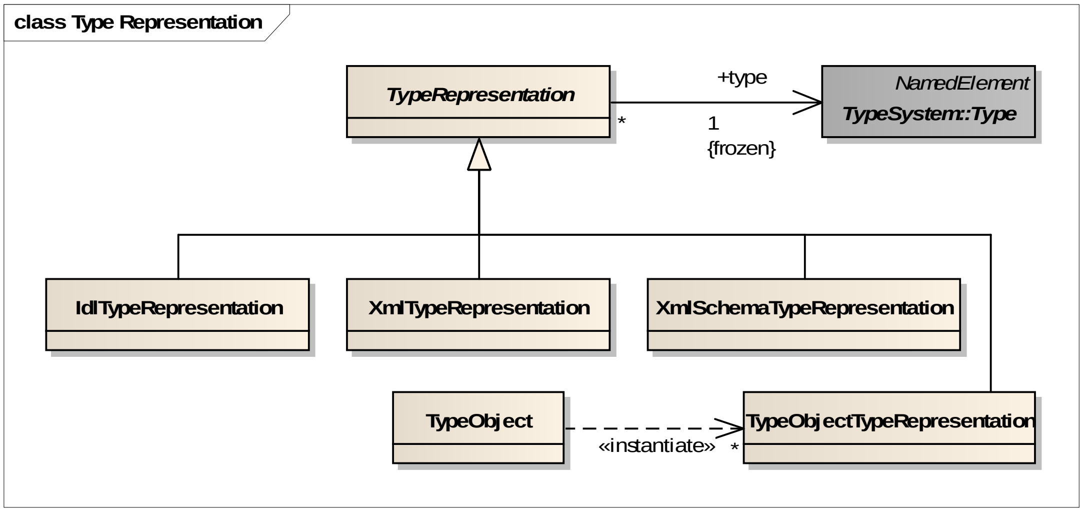

Figure 20 - Type Representation

The Type Representation module specifies the ways in which a type can be externalized so that it may be stored in a file or communicated over the network. Type Representations serve multiple purposes such as:

- Allow a user to describe and document the data type.
- Provide an input to tools that generate code and language-specific constructs to program and manipulate objects of that type.
- Provide an input to tools that want to “parse” and interpret data objects dynamically, without compile-time knowledge of the schema.
- Communicate data types via network messages so that applications can dynamically discover each other’s types or evaluate whether relationships such as *is-assignable-from* are true or false.

This specification introduces multiple equivalent Type Representations. The reason for defining multiple type representations is that each of these is better suited or optimized for a particular purpose. These representations are all equivalent because they describe the same Type System. Consequently, other than convenience or performance, there is no particular reason to use one versus the other.

The alternative representations are summarized in Table 16.

Table 16 - Alternative Type Representations

Type Representation	*Reasons for using it*	*Disadvantages*
IDL	Compact Language. Easy to read and write by humans. Familiar to programmers. Uses constructs close to those in programming languages. Allows re-use of types defined for CORBA. Has standard language bindings to most programming languages.	Perceived as a legacy language by users who prefer XML-based languages. Not as many tools available (parsers, transformations, syntax-aware editors) as XML-languages. Parsing is complex. Requires extensions to support all concepts in the Type System, e.g. keys, optional members, map types, and member IDs.
`TypeObject`	Can provide most compact binary representation. Best suited for communication over a network or as an internal representation of a type.	Not human readable or writable.
XML	Compact XML language. Easy to read and write by humans. Defined to precisely fit the Type System so all concepts (including keys, optional member, etc.) map well. Syntax can be described using XSD allowing the use of editors that assist and verify the syntax of the type. Well-suited for run-time processing due to availability of packages that parse XML.	New language. Based on XML but with a schema that is previously unknown to users.
XSD	Popular standard. Familiar to many users. Human readable. Allows reusing of types defined for other purposes (e.g. web-services). Availability of tools to do syntax checking and editors that assist with auto-completion.	Cumbersome syntax. XSD was conceived as a way to define the syntax of XML documents, not as a way to define data types. No direct support for many of the contructs (e.g keys) or the types in the type model (e.g. arrays, unions, enums), resulting on having to use specific patterns that are hard to remember and error-prone. Very verbose. Hard to read by a programmer.

7.3.1 IDL Type Representation

The type system defined by this specification is designed to allow types to be easily represented using IDL [IDL] with minimal extensions.

7.3.1.1 IDL Compatibility

This specification considers two aspects of IDL compatibility:

- *Backward compatibility with respect to type definitions*: Existing IDL type definitions for use with DDS remain compatible to the extent that those definitions were standards-compliant and based on implementation-independent best practices.
- *Forward compatibility with respect to IDL compilers*: With a few exceptions, IDL type definitions formulated according to this specification will be accepted by IDL compilers that do not conform to this specification.

7.3.1.1.1 Backward Compatibility with Respect to Type Definitions

This specification retains well-established IDL type definition syntax, such as enumeration, structure, union, and sequence definitions.

This specification defines the representation of concepts that were previously represented in implementation-specific ways, such as type inheritance and keys. These representations are defined in subsequent subsections of the IDL Type Representation.

IDL already defines constructs that are orthogonal to the Type System defined by this specification (for example, remote interfaces for use with CORBA). Some DDS users may be using these constructs for implementation-specific purposes or because they use DDS alongside other IDL-based technologies, such as CORBA. These constructs remain legal for use in IDL files provided to IDL compilers compliant with this specification. However, their meanings are undefined with respect to this specification. Compilers that do not support them shall ignore them or issue a warning rather than halting with an error.

7.3.1.1.2 Forward Compatibility with Respect to Compilers

This specification retains well-established IDL type definition syntax, such as enumeration, structure, union, and sequence definitions. This degree of backward compatibility also provides forward compatibility with respect to IDL compilers.

However, this specification also defines new Type System concepts that necessarily had no defined IDL representation, such as maps and annotations. In some cases, such as with annotations, a syntax exists that does not harm compatibility; see section 7.3.1.2.3. In other cases, incompatibility is unavoidable.

The following pragma declarations allow IDL type designers to indicate to their tools and to human readers that their IDL file (or a portion of it) makes use of constructs defined by this specification:

```
#pragma dds_xtopics begin [<version_number>]

// IDL definitions

#pragma dds_xtopics end
```

The optional version number indicates the OMG version number of this specification document. It shall be interpreted without respect to case, and any spaces (for example, in "1.0 Beta 1") shall be replaced with underscores.

For example:

```
#pragma dds_xtopics begin 1.1

struct Base {

   @Key long id;

};
```

```
struct Sub : Base {
};
#pragma dds_xtopics end
```

The above declarations are informative only. The behavior of an IDL compiler upon encountering them is unspecified but may include:

- Silently ignoring them.
- Issuing a warning, perhaps because it does not recognize them, or because it recognizes the pragmas but not the indicated version number.
- Halting with an error, perhaps because it recognized the pragmas and knows that it is not compliant with this specification.

7.3.1.2 Annotation Language

This document defines new kinds of types—for example, bit sets—that cannot be described using existing IDL constructs. It also defines a number of items of meta-data that can be applied to model elements—for example, whether a member of an aggregated type forms part of the key of that type—for which no previous syntax exists in IDL. This document provides a language for describing this extended information and meta-information; this language is extensible by vendors and by users to support future evolution of IDL beyond what is currently envisioned by this specification. The following section defines this facility for defining meta-data annotations and applying those annotations to IDL elements. This facility is based on the similar facility provided by the Java programming language.

7.3.1.2.1 Defining Annotations

Annotation types shall be represented as described in this section. An annotation type is defined by prefixing a local interface definition with the new token "@Annotation," as in the following example:

```
@Annotation
local interface MyAnnotation {
   // ...
};
```

Recall from the Type System Model that annotation types are a form of aggregated type similar to a structure. The members of these types shall be represented using IDL attributes, as shown in the following example:

```
@Annotation
local interface MyAnnotation {
   attribute long my_annotation_member_1;
   attribute double my_annotation_member_2;
};
```

Annotation members have additional constraints that are described above in the Type System Model.

Annotation interface members can take default values; these are expressed by using the keyword "`default`" in between the attribute name and the semicolon, followed by the default value. This value must be a valid IDL literal that is type compatible with the type of the member.

Consider the following example[7]. The `RequestForEnhancement` annotation indicates that a given feature should be implemented in a hypothetical system, and it provides some additional information about the requested enhancement.

```
@Annotation
local interface RequestForEnhancement {
   attribute long id;             // identify the RFE
   attribute string synopsis;     // describe the RFE
   attribute string engineer  default "[unassigned]"; // engineer to implement
   attribute string date      default "[unimplemented]"; // date to implement
};
```

The specified default value may be any legal IDL literal compatible with the declared return type of the method.

7.3.1.2.2 Applying Annotations

Annotations may be applied to any type definition or type member definition. The syntax for doing so is to prefix the definition with an at-sign ('@') and the name of the desired annotation interface. For example:

```
struct Delorean {
   Wheel wheels[4];
   float miles_per_gallon;
   @RequestForEnhancement boolean can_travel_through_time;
};
```

More than one annotation may be applied to the same element, and multiple instances of the same annotation may be applied to the same element.

Annotations can be applied to the implicit discriminator member of a union type by applying them to the discriminator type declaration in the header of the union type's definition:

```
union MyUnion switch (@MyAnnotation long) {
case 0:
   string member_0;
default:
   long default_member;
};
```

As with any IDL identifier, the name of an annotation interface and its members are *not* case-sensitive. To specify multiple annotations, place them one after another, separated by white space.

To specify values for any or all or all of the annotation type's members, follow the name of the annotation interface with parenthesis, and place the member values in a comma-delimited list in between them, where each list item is of the form "*member_name* = *member_value*." Each value must be a compile-time constant. For example:

---

[7] The example annotation type shown is based on one used in the Java annotation tutorial from Sun Microsystems: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

```
struct Delorean {
   @RequestForEnhancement(
      id       = 10,
      synopsis = "Enable time travel",
      engineer = "Mr. Peabody",
      date     = "4/1/3007"
   )
   boolean can_travel_through_time;
};
```

An annotation with an empty list of member values is equivalent to a member list that is omitted altogether.

Any member of the annotation interface may be omitted when the annotation is applied. If a value for a given member is omitted, and that member has a defined default value, it will take that value. If an omitted member does not have a specified default, it will take the default value specified for its type in Section 7.2.2.3.5.5.

If an annotation interface has only a single member, the type designer is recommended to name that member "value." In such a case, the member name may be omitted when applying the annotation. For example:

```
@Annotation
local interface Widget {
   attribute long value;
};

@Widget(5)
struct Gadget {
   // ...
};
```

7.3.1.2.3 Alternative Syntax

It is anticipated that it will take vendors some amount of time to implement this specification. During this time, existing customers may have the need to share IDL files between products that do support this specification and those that do not. In such a case, the extended annotation syntax defined here could be problematic. Therefore, this specification defines an alternative syntax for annotations that will not cause problems for pre-existing IDL compilers.

This alternative syntax uses special comments containing at-signs ('@'), much like the way JavaDoc used "at" comments to attach meta-data to declarations prior to the introduction of an annotation to the Java language. (For example, the conventional way to deprecate a method prior to Java 5 was to place "@deprecated" in the documentation. In Java 5 and above, the preferred way is to use "@Deprecated" in the source code itself, but the JavaDoc-based mechanism is still supported.)

As an alternative to prefixing a declaration with an annotation, it is legal to *follow* the declaration with a single-line comment containing the annotation string. To distinguish such comments from regular comments, there must be no space in between the double slash ("//") and the at-sign ('@'). For example:

```
struct Gadget {
   long my_integer;  //@MyMemberAnnotation("Hello")
}; //@MyTypeAnnotation
```

If multiple annotations are to be applied to the same element, all shall occur on the same line. As before, the at-sign of the first shall *not* be separated from the double slash with any white space; however, subsequent annotations shall be preceded by white space. For example:

```
struct Gadget {
   long my_integer;  //@MyFirstAnnotation(greeting="Hello")  @MySecondAnnotation
}; //@MyTypeAnnotation
```

7.3.1.2.4 Design Rationale (Non-Normative)

The IDL annotation syntax has been designed to closely resemble that of Java in order to appear familiar to Java developers, who represent an important part of the expected user base for this specification. Moreover, the parallels to Java anticipate potential future standard or vendor-specific extensions to this specification without changing core syntax. For example:

- The IDL interface syntax already allows for the expression of multiple inheritance. This specification currently only supports single inheritance. However, if a future extension adds support for multiple inheritance, the annotation definition syntax need not change.
- Java annotations allow for more complex member types—for example, arrays. The syntax for Java array literals could be grafted into this specification by a future extension without breaking existing annotation definitions.

7.3.1.3 Built-in Annotations

This specification defines a number of annotations for use by applications. These types do not appear as annotations at runtime; they exist at runtime only in order to extend the capabilities of IDL. Conformant IDL compilers need not provide actual definitions of these annotations, but must behave as if they did. The equivalent definitions appear below.

7.3.1.3.1 Member IDs

All members of aggregated types have an integral member ID that uniquely identifies them within their defining type. Because IDL has no native syntax for expressing this information, IDs by default are defined implicitly based on the members' relative declaration order. The first member (which, in a union type, is the discriminator) has ID 0, the second ID 1, the third ID 2, and so on.

These implicit ID assignments can be overridden by using the "ID" annotation interface. The equivalent definition of this type follows:

```
@Annotation
local interface ID {
   attribute unsigned long value;
};
```

It is permitted for some members of a type to bear the `ID` annotation while others do not. In such cases, implicit values are assigned in a progression starting from the most-recently specified ID (or an implicit value of zero for the first constant, if there is no previous specified value) and adding one with each successive member.

IDs must be unique within a type and its base types. A non-unique ID is an error.

7.3.1.3.2 Optional Members

By default, a member declared in IDL is not optional. To declare a member optional, apply the "Optional" annotation. The equivalent definition of this type follows:

```
@Annotation
local interface Optional {
   attribute boolean value default true;
};
```

It is an error to declare the same member as both optional and as a key.

7.3.1.3.3 Key Members

By default, members declared in IDL are not considered part of their containing type's key. To declare a member as part of the key, apply the "Key" annotation. The equivalent definition of this type follows:

```
@Annotation
local interface Key {
   attribute boolean value default true;
};
```

It is an error to declare the same member as both optional and as a key.

7.3.1.3.4 Enumerated Constant Values

Prior to this specification, it was not possible to indicate that objects of enumerated types could be stored using an integer size other than 32 bits. This specification provides such a capability using the `BitBound` annotation, which may be applied to enumerated types. It shall have the following equivalent definition:

```
@Annotation
local interface BitBound {
   attribute unsigned short value default 32;
};
```

The `value` member may take any value from 1 to 32, inclusive, when this annotation is applied to an enumerated type.

Furthermore, in IDL, prior to this specification, it was not possible to provide an explicit value for an enumerated constant. The value was always inferred based on the definition order of the constants. That behavior is still supported. However, additionally, this specification allows enumerated constants to be given explicit custom values, just as they can be in the C and C++ programming languages. This can be done by means of the "Value" annotation, which may be applied to individual constants:

```
@Annotation
local interface Value {
   attribute unsigned long value;
};
```

It is permitted for some constants in an enumeration type to bear the `Value` annotation while others do not. In such cases, as in C and C++ enumerations, implicit values are assigned in a progression starting from the most-recently specified value (or an implicit value of zero for the first constant, if there is no previous specified value) and adding one with each successive constant.

7.3.1.3.5 BitSet Types

Bit set types reuse the syntax of IDL enumerations. An enumeration is marked as a bit set with the "`BitSet`" annotation. The bound of the bit set is indicated using the same `BitBound` annotation that may be used with other enumerated types; if it is omitted, the bound of the bit set takes the default value of the `value` member of that annotation. The flags themselves may take default ordinal values, as in a regular enumeration, or may be assigned indexes using the "`Value`" annotation described above. The equivalent definition of the `BitSet` annotation is:

```
@Annotation
local interface BitSet {
};
```

When it annotates a bit set type, the `value` of the `BitBound` can take any value from 1 to 64, inclusive.

An example follows:

```
@BitSet @BitBound(16)
enum MyBitSet {
   FLAG_0,
   FLAG_1,
   @Value(15)
   FLAG_LAST
};
```

Note that it is an error for multiple flags in the same bit set type to have the same index, and therefore it is an error for multiple flags to assign the same value. It is likewise an error to assign any value outside of the range of the bit set type's bound.

7.3.1.3.6 Nested Types

By default, aggregated types and aliases to aggregated types defined in IDL are not considered to be nested types. This designation may be changed by applying the "`Nested`" annotation to a type definition. The equivalent definition of the `Nested` annotation is:

```
@Annotation
local interface Nested {
   attribute boolean value default true;
};
```

7.3.1.3.7 Type Extensibility and Mutability

The extensibility kind of a type may be defined by means of a built-in "Extensibility" annotation. This built-in annotation uses the following enumerated type:

```
enum ExtensibilityKind {
   FINAL_EXTENSIBILITY,
```

```
        EXTENSIBLE_EXTENSIBILITY,
        MUTABLE_EXTENSIBILITY
    };
```

The equivalent definition of the `Extensibility` annotation is:

```
    @Annotation
    local interface Extensibility {
        attribute ExtensibilityKind value;
    };
```

This annotation may be applied to the definitions of aggregated types. It shall be considered an error for it to be applied to the same type multiple times.

In the event that the representation of a given type does not indicate the type's extensibility kind, the type shall be considered extensible. Implementations may provide a mechanism to override this default behavior; for example, IDL compilers may provide configuration options to allow users to specify whether types of unspecified extensibility are to be considered final, extensible, or mutable.

7.3.1.3.8 Must Understand Members

By default, the assignment from an object of type T2 into an object of type T1 where T1 and T2 are non-final types will ignore any members in T2 that are not present in T1. This behavior may be changed by applying the "Must Understand" annotation to a member within a type definition. The equivalent definition of the `MustUnderstand` annotation is:

```
    @Annotation
    local interface MustUnderstand {
        attribute boolean value default true;
    };
```

If the `MustUnderstand` annotation is set to true in particular member M2 of a type T2, then the assignment to an object of type T1 shall fail if the type T1 does not define such a member.

7.3.1.3.9 Verbatim Text

`VerbatimText` objects associated with a constructed type declaration shall be indicated using the following equivalent `Verbatim` annotation:

```
    @Annotation
    local interface Verbatim {
        attribute string<32>    language    default "*";
        attribute string<128>   placement   default "before-declaration";
        attribute string        text;
    };
```

7.3.1.4 Constants and Expressions

IDL allows the declaration of global and namespace-level constant values. It also allows the use of compile-time mathematical expressions, which may include constants, enumeration values, and numeric literals. Such declarations and expressions remain legal IDL. However, they are not reflected directly in the Type System specified here, which assumes that all compile-time-constant values have already been evaluated.

7.3.1.5 Primitive Types

The primitive types specified here directly correlate to the primitive types that already exist in IDL.

Table 17 - IDL primitive type mapping

Type System Model Type	*IDL Type*	*Type System Model Type*	*IDL Type*
`Int16`	`short`	`Float64`	`double`
`UInt16`	`unsigned short`	`Float128`	`long double`
`Int32`	`long`	`Char8`	`char`
`UInt32`	`unsigned long`	`Char32`	`wchar`
`Int64`	`long long`	`Boolean`	`boolean`
`UInt64`	`unsigned long long`	`Byte`	`octet`
`Float32`	`float`		

7.3.1.6 Alias Types

Aliases as described in this specification are fully compatible with the IDL `typedef` construct. The mapping is one-to-one; there is no change necessary.

7.3.1.7 Array and Sequence Types

Arrays and sequences as described in this specification are fully compatible with the IDL constructs of the same names. The mapping is one-to-one; there is no change necessary.

7.3.1.8 String Types

The string container defined by this specification has two element types for which the behavior is defined: `Char8` and `Char32`. Strings of `Char8` shall be represented by the IDL type `string`. Strings of `Char32` shall be represented by the IDL type `wstring`. In either case, any bound shall be retained.

7.3.1.9 Map Types

Map types are an extension to IDL. The syntax is the same as that for sequences with two exceptions:

- The keyword “sequence” is replaced by the new keyword “map.”

- The single type parameter that appears in a sequence definition is replaced by two type parameters in a map definition: the first is the key element type; the latter is the value element type.

For example:

```
map<long, MyModule::MyType> my_member;
```

7.3.1.10 Structure Types

Structures, as defined by this specification, shall be represented using IDL structures having the same name, members, and other properties. The following rules also apply.

7.3.1.10.1 Inheritance

The syntax of IDL structures shall be augmented to allow the expression of single inheritance. Specifically, the production **<struct_type>** from the IDL specification shall be as follows:

<struct_type> ::= "struct" <identifier> [":" <scoped_name>] "{" <member_list> "}"

The **<scoped_name>** production, if present, indicates the name of the structure's super type, which must itself represent a valid Type System structure. If it is not present, the structure has no super type.

Design Rationale (non-normative)

This specification could have leveraged the syntax of IDL valuetypes to express inheritance. However, doing so would have brought its own problems; therefore, that approach was rejected. For example:

- IDL does not permit valuetypes to inherit from structures; however, the distinction between the two is irrelevant to the Type System specified here.
- When serialized in CDR, valuetypes are quite different than structures. They contain type information and other metadata to deal with inheritance and object graphs that are outside of the scope of this specification.
- Valuetypes bring with them a host of other features that are irrelevant to this specification: abstract, truncatable, custom, value boxes, multiple inheritance, and interface support among them.
- Valuetypes are not well supported in all programming languages, such as C.

7.3.1.11 Union Types

Unions as described in this specification are almost fully compatible with the IDL constructs of the same names. The one change is support for additional discriminator types provided by this specification: `Byte` (`octet`) and `Char32` (`wchar`). Compliant IDL parsers shall accept the names of these types in the discriminator position and generate appropriate code according to the appropriate language binding.

7.3.2 XML Type Representation

Types may be defined in an easy-to-read, easy-to-process XML format. This format is defined by an XML schema document (XSD) and a set of semantic rules, which are discussed below.

Design Rationale (non-normative)

The XML Type Representation very much resembles a translation of the grammar of the IDL Type Representation directly into XML. The largest change from such a straightforward translation is that the "built-in annotations" from the IDL Type Representation are here represented as first-class XML constructs—a luxury that is feasible here because this Representation does not predate the definition of the corresponding modeling concepts.

7.3.2.1 Type Representation Management

This Type Representation provides several features that do not directly impact or reflect the Type System. However, they provide capabilities that are necessary or convenient for the organization and management of type declarations. These features are described in this section.

7.3.2.1.1 File Inclusion

As in IDL, files may include other files. Such inclusions shall not be considered semantically meaningful with respect to the Type System Model, but they can be useful as a code maintenance tool.

A file inclusion specified as in this Type Representation shall be considered equivalent to an IDL `#include` of the same file. A formal definition is in "Annex A: XML Type Representation Schema." The following is a non-normative example:

```
<types>
   <include file="my_other_types.xml"/>
</types>
```

Conformant Type Representation compilers need not support the inclusion of files of other Type Representations from within an XML Type Representation document. For example, conformant Type Representation compilers need not support the inclusion of IDL files from XML files.

Design Rationale (non-normative)

XML provides other mechanisms to include one file within another—for example, by defining custom entities. However, these mechanisms cannot provide functionality equivalent to the `#include` of IDL because of when they are interpreted during the XML parsing process.

For example, suppose a type `X` defined in `X.xml` and a type `Y` defined in `Y.xml` both depend on a type `Z` defined in `Z.xml`. Suppose further that an application wishes to use these three types using their Plain Language Bindings in the C programming language. If `X.xml` and `Y.xml` include `Z.xml` using an XML entity definition, this definition will be expanded by the XML parser (upon which the code generator is presumably implemented), and the code generator will never know of the existence of `Z.xml`. It will instead encounter two definitions of `Z`, and the application will fail to build because of multiply defined symbols.

As an alternative, the mechanism described here allows the code generator to *observe* the intention to include `Z.xml` and generate "`#include <Z.h>`," avoiding the multiple definition problem.

7.3.2.1.2 Forward Declarations

As in IDL, C, and C++, a usage of a type must be preceded by a declaration of that type. Therefore, as those languages do, this Type Representation provides for *forward declarations* of types. These declarations are provided for the convenience of code generator implementations; they shall have no representation in the Type Representation Model.

A forward declaration as described in this Type Representation shall be considered semantically equivalent to an IDL forward declaration. A formal definition is in "Annex A: XML Type Representation Schema." The following is a non-normative example:

```
<types>
   <forward_dcl kind="struct" name="MyStructure"/>
</types>
```

7.3.2.1.3 Constants

As in the IDL Type Representation, the XML Type Representation supports declaration of compile-time constant values. Specifically, the string specified in the `value` attribute described below shall have the same syntax as the **<const_exp>** production in the IDL grammar [IDL].

Constants can appear at the top level of a Type Representation file, within a module, or—as in an IDL valuetype—within a structure declaration.

Constants are not reflected directly in the Type System. Instead, mathematical expressions shall be considered to be evaluated at compile time.

The following is a non-normative example:

```
<types>
   <const name="MY_CONSTANT" type="int32" value="2 + 3"/>
</types>
```

7.3.2.2 Basic Types

This Type Representation represents type names with a combination of XML attributes, defined according to the following pattern:

- A "type" attribute, typed by an enumeration `allTypeKind`, indicates whether the type is "basic" (i.e., is a primitive or string)—and if so, which one—or if it is "non-basic" (i.e., any other type).

  Design rationale: As even basic types have identifier names, the use of the `allTypeKind` enumeration does not add to the expressiveness of this Type Representation. However, since primitive types are used frequently, the enumeration allows XML editors to provide context-sensitive completions, improving the user experience.

- A "non-basic type name" attribute indicates the name of the type if it is a non-basic type. It is an error to include this attribute if the `type` attribute does not indicate a non-basic type.

- If the type is a collection type, additional attributes describe its bound(s); see below.

The names of the basic types in this Type Representation have been chosen to resemble terse versions of the corresponding names in the Type System Model.

Table 18 - Primitive and string type names in the XML Type Representation

Type System Model Name	***XML Type Representation Name***
`Boolean`	`boolean`
`Byte`	`byte`
`Char8`	`char8`
`Char32`	`char32`
`Int32`	`int32`
`UInt32`	`uint32`
`Int16`	`int16`

UInt16	uint16
Int64	int64
UInt64	uint64
Float32	float32
Float64	float64
Float128	float128
String<Char8, …>	string
String<Char32, …>	wstring

7.3.2.3 Collection Types

The element type identified by the `type` and `nonBasicTypeName` attributes correspond to the type of a member itself when the member identifies a single value, to the element type when the member is of a sequence or array collection, or to the "value" type of map collection if the member is of a map type. This section and its subsections summarize these rules; the formal grammar can be found in "Annex A: XML Type Representation Schema."

Collection bounds are indicated by attributes named according to the convention `<`*`collection`*`>MaxLength`: `stringMaxLength`, `sequenceMaxLength`, and `mapMaxLength`. The types of these attributes are strings, not integers: the values of these attributes may be any constant expression as defined by the **<const_exp>** production in the IDL grammar [IDL]. The literal expression "-1" shall indicate an unbounded collection; no other "negative" value is permitted.

7.3.2.3.1 String Types

As described above, strings (whether of narrow or wide characters) are considered to be basic types in this Type Representation. Nevertheless, the description of their bounds requires additional attributes.

The `stringMaxLength` attribute, if present, indicates the string's bound. If the attribute is omitted, the string shall be considered unbounded.

The presence of this attribute is legal only when a member's type is a string, a wide string, or an alias to string or wide string. The following examples are non-normative:

```
<struct name="MyStructure">
   <member name="unbounded_string_1"
           type="string"/>
   <member name="unbounded_string_2"
           type="string"
           stringMaxLength="-1"/>
   <member name="bounded_string"
           type="string"
           stringMaxLength="2 + MY_CONSTANT"/>
</struct>
```

7.3.2.3.2 Array Types

The presence of the `arrayDimensions` attribute shall indicate that given member is an array. Array dimensions are represented as a comma-delimited list of dimension bounds in the same order in which those bounds would be given in IDL. Whitespace is allowed around each bound and is not significant.

Compile-time-constant mathematical expressions are also permitted; their syntax shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. As in the IDL Type Representation, such expressions are not expressed directly in the Type System Model but are evaluated first. For example, the following are all valid:

- `arrayDimensions="1"`
- `arrayDimensions="2, MY_CONSTANT + 3"`
- `arrayDimensions=" 6,2, 3 "`

For example:

```
<struct name="MyStructure">
   <member name="my_array_of_42_integers"
           type="int32"
           arrayDimensions="42"/>
</struct>
```

7.3.2.3.3 Sequence Types

The `sequenceMaxLength` attribute, if present, shall indicate that the member is of a sequence type.

The following is a non-normative example:

```
<struct name="MyStructure">
   <member name="my_unbounded_sequence_of_integers"
           type="int32"
           sequenceMaxLength="-1"/>
   <member name="my_bounded_sequence_of_structures"
           type="nonBasic"
           nonBasicTypeName="MyOtherStructure"
           sequenceMaxLength="6 * 3"/>
</struct>
```

7.3.2.3.4 Map Types

Map types must include the following additional information:

- The map's bound, if any, shall be indicated by the `mapMaxLength` attribute. This attribute is required for all map types.
- The type of the map's "key" elements shall be indicated by the `mapKeyType` attribute. This attribute is required for all map types. This attribute is exactly parallel to the `type` attribute (which describes the type of the map's "value" elements): it indicates whether the "key" elements of the map are of a basic or non-basic type and, if basic, which basic

type. If the type is non-basic, the `mapKeyNonBasicTypeName` attribute is also required and is parallel to the `nonBasicTypeName` attribute. If the "key" type is basic, the `mapKeyNonBasicTypeName` attribute is not allowed.

- Only if the map's "key" type is a string type, the attribute `mapKeyStringMaxLength`, if present, shall indicate the bound of that string type. If the "key" type is a string type, and this attribute is omitted, the string shall be considered unbounded. If the "key" type is not a string type, this attribute is not allowed.

The following is a non-normative example:

```
<struct name="MyStructure">
   <member name="my_unbounded_maps_of_integers_to_floats"
           type="int32"
           mapKeyType="float32"
           mapMaxLength="-1"/>
   <member name="my_bounded_map_of_strings_to_structures"
           mapKeyType="string"
           mapKeyStringMaxLength="128"
           type="nonBasic"
           nonBasicTypeName="MyOtherStructure"
           mapMaxLength="6 * 3"/>
</struct>
```

7.3.2.3.5 Combinations of Collection Types

A type may be a sequence of arrays, a map of strings to sequences, or some other complex combination of collection types. It's therefore important to understand, if some *combination* of `stringMaxLength`, `sequenceMaxLength`, and `mapMaxLength` are present, which takes precedent. The following list is ordered from most-tightly-binding to least-tightly-binding:

1. String designations, including `stringMaxLength`
2. Sequence designations, including `sequenceMaxLength`
3. Array designations, including `arrayDimensions`
4. Map designations, including `mapMaxLength`.

To indicate a type composed in a different order (for example, a sequence of arrays), it is necessary to interpose an alias definition.

For example, a member specifying all of these would define a map whose values are arrays of sequences of strings. Further examples follow:

```
<struct name="MyStructure">
   <member name="my_array_of_strings"
           type="string"
           stringMaxLength="-1"
           arrayDimensions="20"/>
```

```
    <member name="my_array_of_sequences_of_integers"
            type="int32"
            sequenceMaxLength="6 * 3"
            arrayDimensions="20"/>
</struct>
```

7.3.2.4 Aggregated Types

Aggregated types include those types that define internal named members taking per-instance values: annotations, structures, and unions.

The Type System defines a number of properties for aggregated types and their members:

- `extensibility_kind`
- `nested`
- `key`
- `optional`
- `must_understand`, etc.

The IDL Type Representation is based on IDL, which provides no syntax to provide values for these attributes; therefore, that Type Representation makes use of built-in annotations for this purpose. In contrast, the XML Type Definition is able to express these properties directly.

For example, structures and unions may indicate whether they are extensible/mutable and/or nested types:

```
<struct name="MyStructure"
        extensibility="mutable"
        nested="true">
   …
</struct>
```

In the event that the representation of a given type does not indicate the type's extensibility kind, an implementation may make its own determination. In particular, type representation compilers shall provide configuration options to allow users to specify whether types of unspecified extensibility will be considered final, extensible, or mutable.

7.3.2.4.1 Annotations

There are two primary declarations pertaining to annotations: annotation types and the applications of them to types and type members, specifying values for the annotation's own members.

The following is a non-normative example:

```
<annotation name="MyAnnotation">
    <member name="widgets"
            type="int32"/>
</annotation>
```

```
<struct name="MyStructure">
   <annotate name="MyAnnotation">
      <member name="widgets" value="5"/>
   </annotate>
   …
</struct>
```

7.3.2.4.2 Structures

Structures contain four kinds of declarations:

- Applied annotations
- Verbatim text
- Members
- Constants

Constants and applied annotations are described above. The other elements are described in the sections below.

7.3.2.4.2.1 Verbatim Text

As described in Section 7.2.2.3.7, types may store blocks of text to be used by Type Representation compilers. These are represented within a structure's declaration as shown in the following non-normative example:

```
<struct name="MyStructure">
   <verbatim language="Java" placement="before-declaration">
      /**
       * This is a JavaDoc comment.
       */
   </verbatim>
   …
</struct>
```

7.3.2.4.2.2 Members

Each structure type shall include one or more members. Each member of a structure type can indicate individually whether or not it is a key member and whether or not it is an optional member.

```
<struct name="structMemberDecl">
   <member name="my_key_field"
           type="int32"
           key="true"
           optional="false"/>
</struct>
```

7.3.2.4.2.3 Inheritance

A structure declaration's `baseType` attribute indicates the name of the structure's base type, if any; if it is omitted, then the structure has no base type. For example:

```
<struct name="MyStructure" baseType="MyOtherStructure">
   …
</struct>
```

7.3.2.4.3 Unions

In addition to the `annotate` and `verbatim` elements they share with other aggregated types (see above), unions contain two kinds of members: exactly one discriminator member (identified by a `discriminator` element) and one or more cases (identified by `case` members). The discriminator member must be declared before the others.

Each case of a union contains one or more discriminator values (`caseDiscriminator` elements) and one data `member`. A case discriminator is a string expression, the syntax of which shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. The literal "default" is also allowed; it indicates that the corresponding case is the default case—there can only be one such within a given union declaration.

For example:

```
<union name="MyUnion">
   <discriminator type="int32"/>
   <case>
      <caseDiscriminator value="1"/>
      <caseDiscriminator value="2"/>
      <member name="small_value" type="float32"/>
   </case>
   <case>
      <caseDiscriminator value="default"/>
      <member name="large_value" type="float64"/>
   </case>
</union>
```

The example above is equivalent to the following IDL type:

```
union MyUnion switch (long) {
   case 1:
   case 2:
      float small_value;
   default:
      double large_value;
};
```

7.3.2.5 Aliases

Alias definitions are defined in `typedef` elements. They have syntax very similar to that of structure members.

For example:

```
<typedef name="MyAliasToSequenceOfStructures"
         type="nonBasic"
         nonBasicTypeName="MyStructure"
         sequenceMaxLength="16"/>
```

7.3.2.6 Enumerations

Enumerated types consist of a list of “enumerator” constants, each of which has a name and a value. The syntax of the value shall be defined by the **<const_exp>** production in the IDL grammar [IDL]. If the value is omitted, it shall be assigned automatically.

For example:

```
<enum name="MyEnumeration" bitBound="16">
   <enumerator name="CONSTANT_1" value="0"/>
   <enumerator name="CONSTANT_2" value="0+1"/>
   <enumerator name="CONSTANT_3"/>
</enum>
```

7.3.2.7 Bit Sets

A bit set type defines a sequence of flags, each of which shall identify one of the bits in the bit set.

For example:

```
<bitset name="MyBitSet" bitBound="64">
   <flag name="FIRST_BIT" value="0"/>
   <flag name="SECOND_BIT" value="1"/>
</bitset>
```

7.3.2.8 Modules

A module groups type declarations and serves as a namespace for those definitions.

```
<module name="MyModule1">
   <struct name="MyStructure">
      <member name="my_member" type="int64"/>
   </struct>
</module>

<module name="MyModule2">
   <struct name="MyStructure">
```

```
        <member name="my_member"
                type="nonBasic"
                nonBasicTypeName="MyModule1::MyStructure"/>
    </struct>
</module>
```

7.3.3 XSD Type Representation

Types can be defined using an XML schema document (XSD). The format is based on the standard IDL-to-XSD mapping [IDL-XSD]. An XSD Representation of a given type shall be as if the OMG-standard IDL-to-XSD mapping were applied to the IDL Representation of the type as defined in Section 7.3.1. That mapping is augmented as follows to address IDL extensions defined by this specification. The resulting XSD representation may be embedded within a WSDL file or may occur as an independent XSD document.

7.3.3.1 Annotations

It is possible to both define and apply annotations using the XSD Type Representation; these tasks shall be accomplished using XSD Annotations. (To avoid confusion, for the remainder of this section, an annotation as defined by the Type System Model in this document will be referred to as an "OMG Annotation." An annotation as defined by the XML Schema specification shall be referred to as an "XSD Annotation.")

7.3.3.1.1 Defining Annotation Types

OMG Annotation types shall be defined using XSD-standard `complexType` definitions. Any `complexType` definition immediately containing an XSD Annotation with an `appInfo` element having a `source` attribute value of `http://www.omg.org/Type/Annotation/Definition` shall be considered to be an OMG Annotation. Such `complexType` definitions, henceforth referred to as "Annotation `complexType` Definitions" shall conform to the structure defined in this section.

Each attribute of an Annotation `complexType` Definition shall define a member of the corresponding OMG Annotation type:

- The `name` of the attribute shall specify the name of the OMG Annotation type member.
- The `type` of the attribute shall specify the name of the type of the OMG Annotation type member.
- A `default` value, if present, shall specify the default value of the OMG Annotation type member.

The meanings of any sub-elements defined for an Annotation `complexType` Definition are unspecified. The following example provides equivalent definitions for an OMG Annotation type in both IDL and XSD.

IDL	***XSD***

`@Annotation`
`local interface MyAnnotation`
`{`
`  attribute long widgets;`
`  attribute double gadgets`
	`<xsd:complexType name="MyAnnotation">`
	`  <xsd:annotation>`
	`    <xsd:appInfo source=`
	`      "http://www.omg.org/Type/Annotation/Definition"/>`
	`  </xsd:annotation>`

`default 42.0;`
 	
`};`
	`<xsd:attribute name="widgets"`
	`type="xsd:int"/>`
	`<xsd:attribute name="gadgets"`
	`type="xsd:double"`
	`default="42.0"/>`
	`</xsd:complexType>`

Figure 21 - XSD annotation example

7.3.3.1.2 Applying Annotations

OMG Annotations shall be applied to a definition by declaring, immediately within that definition's XML element, an XSD Annotation containing an `appInfo` with its `source` attribute set to `http://www.omg.org/Type/Annotation/Usage`. The structure of such an `appInfo` element shall conform to that defined in this section.

The `appInfo` element shall contain an element `annotate` for each OMG Annotation to be applied. For syntactic validation purposes, the definition of the `annotate` element shall be as follows:

```
<xsd:schema targetNamespace="http://www.omg.org/Type"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="annotate">
      <xsd:attribute name="type" type="xs:string" use="required"/>
      <xsd:anyAttribute processContents="skip"/>
   </xsd:complexType>
   …
</xsd:schema>
```

However, for semantic validation purposes, the `annotate` element shall contain attribute values corresponding to any subset of the attributes defined by the OMG Annotation type indicated by its required `type` attribute.

In the following example, the OMG Annotation `MyAnnotation` defined in the previous example is applied to a structure definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:omg="http://www.omg.org/Type"
            xmlns:tns="http://www.omg.org/IDL-Mapped/"
            targetNamespace="http://www.omg.org/IDL-Mapped/">
   <xsd:complexType name="MyStructure">
      <xsd:annotation>
         <xsd:appInfo source="http://www.omg.org/Type/Annotation/Usage">
            <omg:annotate omg:type="MyAnnotation"
                          widgets="12"
```

```
                                  gadgets="75.0"/>
                </xsd:appInfo>
            </xsd:annotation>
        </xsd:complexType>
    </xsd:schema>
```

7.3.3.1.3 Built-in Annotations

Unless otherwise noted, those Type System concepts represented with built-in annotations in the IDL Type Representation shall be represented by equivalent built-in annotations in this Type Representation.

7.3.3.2 Structures

The representations of structures and their members shall be augmented as described below.

7.3.3.2.1 Inheritance

The subtype shall extend its base type using an XSD `complexContent` element. For example, the following types in the IDL Type Representation and XSD Type Representation are equivalent:

IDL	***XSD***

struct MyBaseType {
   long inherited_member;
};
 	
struct MyExtendedType : MyBaseType {
   long new_member;
};
	<xs:complexType name="MyBaseType">
	    <xs:sequence>
	       <xs:element name="inherited_member"
	                   type="xs:int"/>
	    </xs:sequence>
	</xs:complexType>
	
	<xs:complexType name="MyExtendedType">
	    <xs:complexContent>
	       <xs:extension base="MyBaseType">
	          <xs:sequence>
	             <xs:element name="new_member"
	                         type="xs:int"/>
	          </xs:sequence>
	       </extension>
	    </xs:complexContent>
	</xs:complexType>

Figure 22 - XSD structure inheritance example

7.3.3.2.2 Optional Members

Optional members of an aggregated type shall be indicated with a `minOccurs` attribute value of 0 instead of 1. For example:

```
<xsd:complexType name="MyType">
   <xsd:sequence>
      <xsd:element name="my_int" minOccurs="0" maxOccurs="1" type="xsd:int"/>
   </xsd:sequence>
</xsd:complexType>
```

7.3.3.3 Nested Types

For each type T that is *not* a nested type, the schema shall define an XML element of that type suitable for use as a document root. The name of this element shall be the fully qualified name of T.

For example, for the structure "`MyStructure`" in the module "`MyModule`" (named "`MyModule.MyStructure`" in this Type Representation) the schema shall include a declaration like the following:

```
<xs:element name="MyModule.MyStructure" type="MyModule.MyStructure"/>
```

7.3.3.4 Maps

A map declaration is superficially like a structure declaration; however, the XSD `sequence` declaration specifies a `maxOccurs` multiplicity equal to the bound of the map (or `unbounded` if the map is unbounded). The map elements are represented by elements named `key` and `value`, each of which must occur exactly once for each iteration of the sequence.

For example, the following is a map of integers to floating-point numbers with a bound of 32:

```
<xsd:complexType name="MyMap">
   <xsd:sequence maxOccurs="32">
      <xsd:element name="key" minOccurs="1" maxOccurs="1" type="xsd:int"/>
      <xsd:element name="value" minOccurs="1" maxOccurs="1" type="xsd:double"/>
   </xsd:sequence>
</xsd:complexType>
```

7.3.4 Representing Types with TypeObject

Any type can be described using the "meta"-type below, which can be serialized using any data representation.

The `TypeObject` type is a type defined according to the type system defined by this specification. It is designed to describe other types in that type system; in that sense, it is a *meta*-type. It is therefore somewhat different than the other type representations defined by this specification: it is not a type representation itself; rather, *data* representations of objects of the `TypeObject` type are *type* representations for other types. `TypeObject` is designed to provide compact representations for types that are suitable for embedding within data objects such as can be described by this specification[8].

See "Annex B: Representing Types with TypeObject" for the formal definition of the `TypeObject` type.

[8] For example, `TypeObject` objects are used to propagate type information within the DDS built-in topics; see section 7.6.2. Samples of these topics are conventionally represented using the CDR Data Representation [RTPS].

7.3.4.1 Overview

Types and the modules that contain them are stored in "Type Libraries." A TypeObject object contains (*a*) a single TypeLibrary and (*b*) (optionally) identifies a single type within that library.

7.3.4.1.1 References Among Types

Rather than refer to one another by name, as in some other Type Representations, types within this Type Representation they refer to one another by a 32-bit integral "type ID" for the sake of compactness. This ID has no representation in the Type System Model, nor does its value need to be the same from one serialization of a given set of types to another. Type ID values are strictly artifacts of a given serialization of this Type Representation to allow types to refer to one another within that serialization. For example, an implementation may assign type IDs serially, and since the collection of types within a module is unordered, it may serialize types in different orders from run to run, resulting in different assignments of ID values to types.

To allow types to refer to one another unambiguously, a given TypeId value shall uniquely identify a type within the TypeLibrary contained by a TypeObject object and in any other Type Libraries contains recursively therein. It shall have no narrower scope. There is no restriction that a type's definition must occur a TypeId reference to it; there is no concept of a forward declaration in this Type Representation.

7.3.4.1.2 Type Hierarchy

For each type kind, there exists in this Type Representation a structure to describe types of that kind; each of these is named for its type kind followed by the suffix "Type" (for example, "ArrayType," "StructureType," etc.).

The type hierarchy defined by the Type System Model is reflected here. At its root is the type Type, which combines both Type and ConstructedType from the Type System Model. This base type provides the following data, common to all types:

- The type's extensibility. The extensibility kind is represented by two bits in a "flag" bit set: IS_FINAL and IS_MUTABLE. A '1' in the former and a '0' in the latter indicate a final type. A '0' in both indicates an extensible type. A '0' in the former and a '1' in the latter indicate a mutable type. The meaning of a '1' in both is unspecified.
- Whether or not the type is a nested type. This property is indicated by a bit in the "flag" bit set: IS_NESTED.
- The type's name.
- The type's TypeId (explained above).
- Any annotations applied to the type.

The member IDs of mutable types are defined in enumerations whose names end in "MemberId"; for example, the member IDs of the StructureType type are defined in the enumeration StructureTypeMemberId. (These enumerations avoid the use of "magic numbers" in the type definitions.) By convention in this Type Representation, the numeric values of the constants defined by these enumerations increases by 100 which each step in the type hierarchy in order to leave room for further evolution of this specification. For example, The member IDs in the Type base structure are in the range 0, 1, …. Those in the StructureType structure, which extends Type, are in the range 100, 101, ….

7.3.4.2 Primitive Types

Primitive types are indicated by TypeId values, just as are constructed types. Because the definitions of the primitive types are not included in the TypeLibrary, the TypeId values for these types are predefined. The assigned ID values for constructed types must not overlap with these predefined values.

7.3.4.3 Collection Types

The structure `CollectionType` is the base type for all collection types. It identifies the common element type of the collection.

7.3.4.3.1 String Types

The structure `StringType` describes a string type. Its element type indicates whether it is a string of narrow or wide characters. It also identifies the string type's bound; a bound of zero indicates an unbounded string.

7.3.4.3.2 Array Types

The structure `ArrayType` describes an array type. It contains a sequence of bound values, one for each of its dimensions.

7.3.4.3.3 Sequence Types

The structure `SequenceType` describes a sequence type. It identifies the sequence type's bound; a bound of zero indicates an unbounded sequence.

7.3.4.3.4 Map Types

The structure `MapType` describes a map type. The element type it inherits from `CollectionType` identifies the type of the map's "value" elements. A further member, `key_element_type`, identifies the type of the map's "key" elements.

`MapType` also identifies the map type's bound; a bound of zero indicates an unbounded map.

7.3.4.4 Aggregated Types

Aggregated types are those types, the objects of which contain an ordered collection of heterogeneous values, its members. A member of an aggregated type is represented by the structure `Member`. This structure contains the information common to all members, such as the members name and type, whether or not it is optional, whether or not it is a key, etc.

The different kinds of aggregated types store slightly different information along with their members. For example, annotation members may take custom default values, and union members are associated with case labels. Therefore, these kinds of types are associated with their own `Member` sub-types, described below.

7.3.4.4.1 Annotations

There are two aspects to annotations: annotation types, which are represented by instances of the `AnnotationType` structure, and the application of those annotation types to other types and their members; the latter are represented by instances of the `AnnotationUsage` structure.

`AnnotationType` extends `Type` and contains a sequence of annotation members. These latter are represented by instances of the `AnnotationMember` structure, which extends `Member` with the addition of a default value. `AnnotationType` also identifies the base type of the annotation, if any.

An `AnnotationUsage` associated a concrete literal value with the members of an annotation type. As such, it identifies that type (with a `TypeId`) and contains a sequence of member values, which identify the annotation members whose values they set by member ID.

7.3.4.4.2 Structures

Structure types, represented by instances of the `StructureType` structure, do not associate additional data with their members. Therefore `StructureType` composes `Member` directly; there is no `Member` subtype corresponding to structures.

Structure types also identify their base type, if any.

7.3.4.4.3 Unions

The UnionType structure extends Type and contains a sequence of union members. These latter are represented by instances of the UnionMember structure, which extends Member with the addition of a sequence of case labels. The default label, since it has no value, does not appear in this list. Instead, it is indicated, if present, by a per-member IS_UNION_DEFAULT_MEMBER flag.

UnionType does not explicitly distinguish its discriminator member from its other members; doing so would be unnecessary, since by definition the discriminator is the first member and the only non-optional one.

7.3.4.5 Aliases

The structure AliasType describes an alias type. It identifies the base type of the alias.

7.3.4.6 Bit Sets

The structure BitSetType describes a bit set type. It contains a sequence of objects of type Bit, each of which contains the name and index of an identified bit within the bit set. Reserved bits are not represented.

7.3.4.7 Modules

The structure Module describes a module. In addition to its name, it contains a TypeLibrary, just as the TypeObject structure does.

7.4 Data Representation

The Data Representation module specifies the ways in which a data object of a given type can be externalized so that it can be stored in a file or communicated over the network. This is also commonly referred as “data serialization” or “data marshaling.”

Data Representations serve multiple purposes such as:

- Represent data in a “byte stream” so it can be sent over the network
- Represent data in a “byte stream” so it can be stored in a file
- Represent data in a human-readable form so it can be displayed to the user
- Provide a language for the user to enter data-values to a tool or specify them in a file

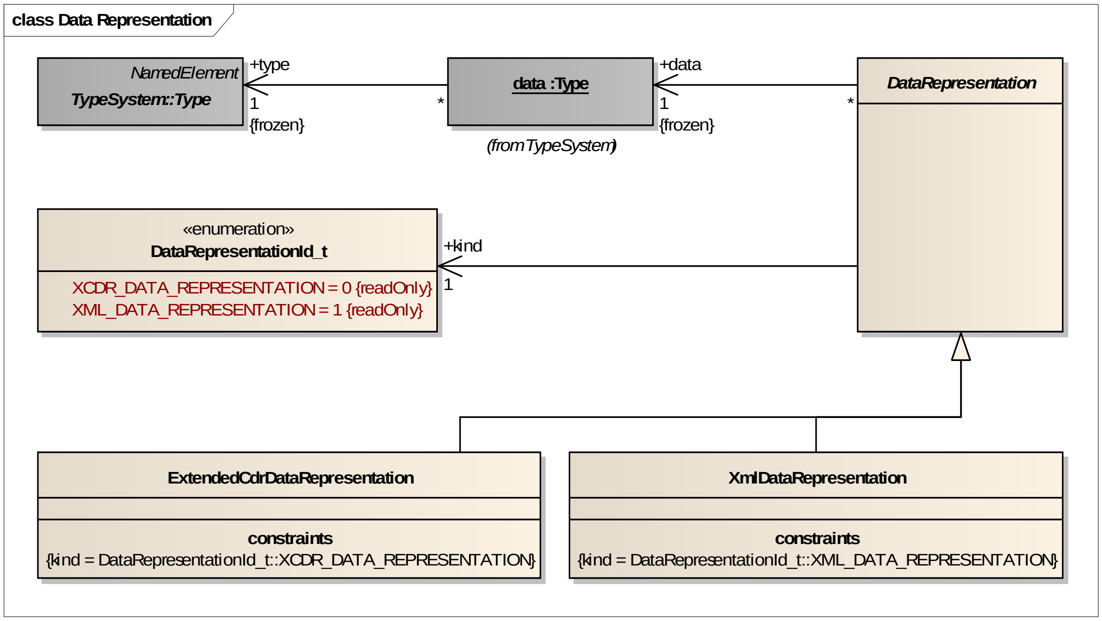

Figure 23 - Data Representation—conceptual model

This specification introduces multiple Data Representations. The reason for defining multiple type representations is that each of these is better suited or optimized for a particular purpose. These representations are all mostly equivalent. Consequently, other than convenience or performance, there is little reason to use one versus the other.

The alternative representations are summarized in Table 19.

Table 19 - Alternative Data Representations

Data Representation	***Reasons for using it***	***Disadvantages***
Extended CDR, encompassing both "traditional" CDR and parameterized CDR	Compact and efficient binary representation. Minimizes CPU and Bandwidth used. Supports type evolution. Existing international OMG Standard. (Traditional CDR from CORBA [CDR]; parameterized CDR from RTPS [RTPS].) Already in used in the DDS Interoperability Protocol.	Not human readable.
XML	Human Readable Easily parsed and transformed with standard tools	CPU Intensive Uses 10 or 20 times more space than CDR

7.4.1 Extended CDR Data Representation

This specification defines an extension of the OMG CDR representation [CDR] that is able to accommodate both optional members and extensible/mutable types:

- The specification leverages the OMG CDR representation for all primitive types and non-mutable constructed types where the CDR representation is well defined.

- The specification introduces extensions to handle optional members, bit sets, and maps.
- The specification leverages the RTPS Parameter List representation [RTPS] to handle type extensibility.

7.4.1.1 Use of the (Traditional) OMG CDR Representation

The traditional CDR representation shall be used for final and extensible types, including (trivially) primitive types. It shall also be used for all string, sequence, and map types. Aggregated types declared as mutable shall use the Parameterized CDR representation described in Section 7.4.1.2.

The CDR representation is based on the CDR representation format [CDR] with the minimal extensions described below needed to handle the new types and concepts introduced by this specification

7.4.1.1.1 Character Data

Objects of `Char8` and `String<Char8>` types shall be represented using the ISO-8859-1 character encoding.

Objects of `Char32` and `String<Char32>` types shall be represented using the UTF-32 character encoding. (While verbose, the encoding uses fixed-width characters and is thus amenable to rapid processing.)

7.4.1.1.2 Enumeration Types

Objects of enumeration types shall be serialized as integers, the sizes of which shall depend on the “bit bound” of their associated type.

Table 20 - Serialization of enumeration types

Corresponding Integer Type	***Bit Bound***
`Int16`	1-16
`Int32`	17-32 (32 bits is the default size, and corresponds to all enumeration types prior to this specification)

7.4.1.1.3 BitSet Types

Objects of bit set types shall be serialized in the same way as the following primitive types, depending on the bit set’s bound:

Table 21 - Serialization of bit set types

Bound	***Corresponding Primitive Type***
[1..8]	`Byte`
[9..16]	`UInt16`
[17..32]	`UInt32`
[33..64]	`UInt64`

Bit indexes are counted from zero starting at the least-significant bit of the full byte size of the bit set. In the case where the bound of the bit set is less than the number of bits in the corresponding primitive type, the states of the remaining serialized bits are not specified, and those bits are not considered to be part of the bit set.

7.4.1.1.4 Map Types

Objects of map types shall be represented according to the following equivalent IDL2:

```
struct MapEntry_<key_type>_<value_type>[_<bound>] {
   <key_type> key;
   <value_type> value;
};

typedef sequence<MapEntry_<key_type>_<value_type>[_<bound>][, <bound>]>
Map_<key_type>_<value_type>[_<bound>];
```

The *<key_type>* and *<value_type>* names are as defined the Type System. See also Section 7.2.2.3.4, which defines the implicit names of collection types.

For example, objects of the following IDL map type:

```
map<long, float>
```

…shall be serialized as if they were of the following IDL sequence type:

```
struct MapEntry_Int32_Float32 {
   long key;
   float value;
};

typedef sequence<MapEntry_Int32_Float32> Map_Int32_Float32;
```

7.4.1.1.5 Structures

Objects of structure type shall be represented as defined by the CDR specification [CDR], augmented as described below.

7.4.1.1.5.1 Inheritance

The members defined by the base type, if any, shall be serialized before the members of their derived types. The representation shall be exactly as if all of the members had been defined, in the same order, in the most-derived type.

7.4.1.1.5.2 Optional Members

Structure members marked as optional shall be represented by inserting the representation of a `UInt32` ahead of the representation of the member. The value of the `UInt32` shall contain the length in `Bytes` of the representation of the member that follows. If the member is not present the value of the `UInt32` shall be zero.

This data representation does not support the omission of non-optional members, nor does it support member reordering. All non-optional members must be represented, and all represented members must be in the order in which they were defined in the type definition.

7.4.1.2 Parameterized CDR Representation

The parameterized CDR representation is based on the RTPS Parameter List CDR data representation [RTPS]. It shall be used to represent any object belonging to a mutable aggregated type.

The RTPS Parameter List CDR representation defines a parameter list data structure, in which each parameter is identified by a two-byte parameter ID, which is followed by a two-byte parameter length and finally the parameter data itself. One parameter follows another until a list-terminating sentinel is reached. This data representation reuses that data structure: an object of an aggregated type corresponds to a parameter list, and each of its members corresponds to a single parameter of that list.

7.4.1.2.1 Interpretation of Parameter ID Values

As described in section 9.6.2.2.1, *ParameterId space*, of the RTPS Specification, the 16-bit-wide parameter ID range may be interpreted as a two-bit-wide bit set followed by a 14-bit wide unsigned integer.

- The first bit of the bit set—the most-significant bit of 16-bit-wide the parameter ID as a whole—indicates whether the parameter has an implementation-specific interpretation. This specification refers to this bit as `FLAG_IMPL_EXTENSION`.
- The second bit of the bit set indicates whether the parameter, if its ID is not recognized by the consuming implementation, may be simply ignored or whether it causes the entire data sample to be discarded. This specification refers to this bit as `FLAG_MUST_UNDERSTAND`. This bit shall be set if and only if the `must_understand` property of the member being encapsulated is set to true.

Within the 14-bit-wide integer region of the parameter ID, this specification further reserves the largest 255 values—from 16,129 (0x3F01) to 16,383 (0x3FFF)—for use by the OMG in this specification and future specifications. The following table identifies the reserved parameter ID values.

Table 22 - Reserved parameter ID values

Name	*14-Bit Hex Value(s)*	*Description*
`PID_EXTENDED`	`0x3F01`	Allows the specification of large member ID and/or data length values; see below
`PID_LIST_END`	`0x3F02`	Indicates the end of the parameter list data structure. RTPS specifies that the PID value 1 shall be used to terminate parameter lists within the DDS built-in topic data types. Rather than reserving this parameter ID for all types, thereby complicating the member ID-to-parameter ID mapping rules for all producers and consumers of this Data Representation, *Simple Discovery types only* shall be subject to a special case: member ID 1 shall not be used, and either parameter ID 0x3F02 or parameter ID 1 shall terminate the parameter list.
`PID_IGNORE`[9]	`0x3F03`	All consumers of this Data Representation shall ignore parameters with this ID.
Reserved for OMG	`0x3F04-0x3FFF`	*Reserved for OMG*

[9] **Design rationale (non-normative)**: RTPS uses PID 0 (“PID_PAD”), corresponding to member ID 0, as a padding field. PID_IGNORE applies this concept to all data types using this Data Representation. The additional reservation of PID 0 is not necessary: because the types defined by RTPS do not use member ID 0, consumers of those types will naturally ignore any incidence of its corresponding PID that they encounter.

This specification extends the parameter list data structure to permit 32-bit parameter IDs and 32-bit data sizes. This extension uses the reserved 16-bit parameter ID `PID_EXTENDED`. The length of this parameter shall be at least eight bytes: the first four bytes of the parameter data shall be interpreted as a set of four reserved bit flags followed by the 28-bit member ID; the second four bytes shall be interpreted as a 32-bit unsigned data length measured from the end of that field until the start of the next 16-bit parameter ID. If the 16-bit length is greater than eight, the additional contents are undefined and are reserved for future use by OMG specifications.

The setting of the `FLAG_MUST_UNDERSTAND` bit in the 16-bit parameter ID shall be interpreted to apply to the extended parameter as well, not just to the 12 bytes of the `PID_EXTENDED` parameter itself. That is, if the implementation decides to skip the parameter, it must skip the entire data length described by the 32-bit data length field. (If it does not, it could incorrectly start parsing the 32-bit data as if it contained nested parameters, which may or may not be correct.)

7.4.1.2.2 Member ID-to-Parameter ID Mapping

The mapping from member IDs to parameters shall be as follows:

- Member IDs from 0 to 16,128 (0x3F00) inclusive shall be represented exactly in the lower 14 bits of the parameter ID.
- All other member IDs must be expressed using the extended parameter header format.
- Almost any parameter can legally be expressed using extended parameter headers. There is no requirement that parameters that *could* be described with the shorter header defined by the RTPS Specification *must* be described that way; if a parameter could be described using a short parameter header or an extended header, the short and extended expressions of that header shall be considered totally equivalent.

7.4.1.2.3 Omission and Reordering of Members of Aggregated Types

Because each parameter (type member, in this case) is explicitly identified, and identification of structure members occurs based on the IDs of those parameters, members of structures may appear in any order. Furthermore, any structure member's value may be omitted. In such a case, if the member is not optional, it logically takes its default value. If the member is optional, it takes no value at all.

Because union members are identified based on a discriminator value, the value of the discriminator member must be serialized before the value of the current non-discriminator member. Neither member value may be omitted.

7.4.1.2.4 Nested Objects

In the case where an object of an aggregated mutable type contains another object of an aggregated mutable type, one parameter list will contain another. In that case, parameter IDs are interpreted relative to the innermost type definition. (For instance, a type `Foo` may contain an instance of type `Bar`. Both `Foo` and `Bar` may define a member with ID 5. Inside the parameter list corresponding to the `Bar` object, an occurrence of parameter ID 5 shall be considered to refer to `Bar`'s member 5, not to `Foo`'s member 5.)

Likewise, an occurrence of `PID_LIST_END` indicates the conclusion of the innermost parameter list.

7.4.2 XML Data Representation

The XML Data Representation provides for the serialization of individual data samples in XML.

Each data sample shall constitute a separate XML document. The structure of that document shall conform to the XML Schema Type Representation for the sample's corresponding type definition.

(Note that, unlike in the CDR Data Representation, samples of mutable types are serialized no differently than samples of final or extensible types.)

For example, objects of the (non-normative) “MyStructure” structure, defined in IDL below…

```
@Nested
struct MyInnerStructure {
   long my_integer;
};

struct MyStructure {
   MyInnerStructure inner;
   sequence<double> my_sequence_of_doubles;
};
```

…would be represented like this:

```
<MyStructure>
   <inner>
      <my_integer>5<my_integer>
   </inner>
   <my_sequence_of_doubles>
      <item>10.0</item>
      <item>20.0</item>
      <item>30.0</item>
   </my_sequence_of_doubles>
</MyStructure>
```

7.5 Language Binding

The Language Binding Module specifies the alternative programming-language mechanisms an application can use to construct and introspect types as well as objects of those types. These mechanisms include a Dynamic API that allows an application to interact with types and data without compile-time knowledge of the type.

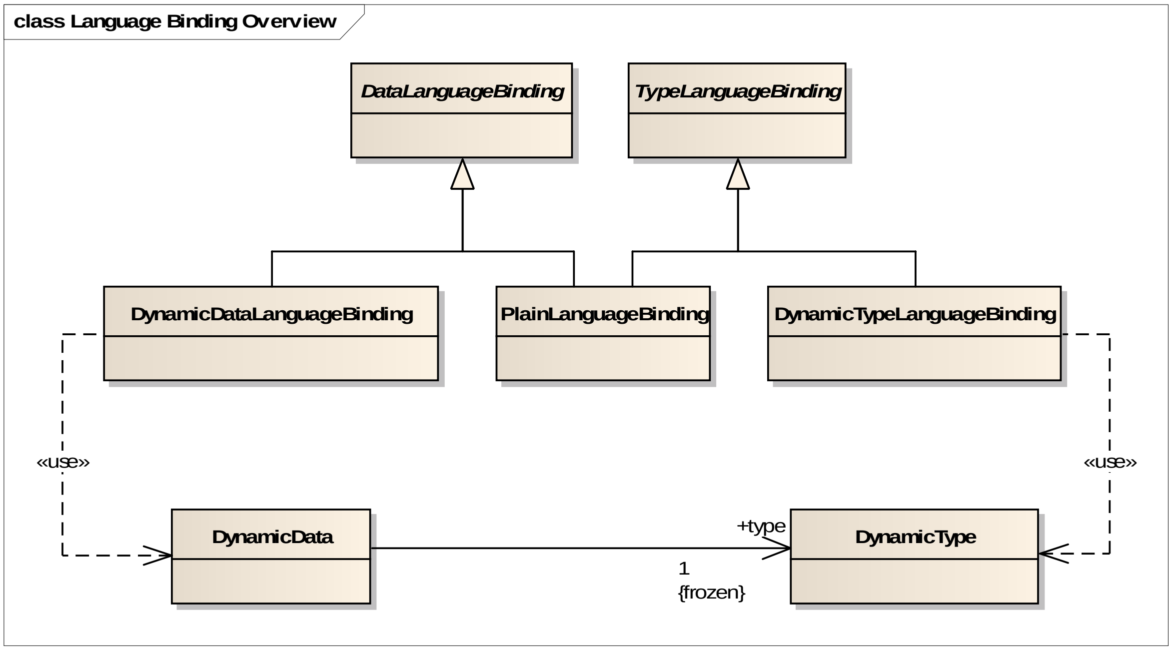

Figure 24 - Language Bindings—conceptual model

The specification defines two language bindings: Plain Language Objects and Dynamic Data. The main characteristics and motivation for each of these binding are described in Table 23.

The Type Language Binding provides an API to manipulate types. This includes constructing new types as well as introspecting existing types. The API is the same regardless of the type, allowing applications to manipulate types that were not known at compile time. This API is similar in purpose to the `java.lang.Class` class in Java.

The principal mechanism to interact with a Type is the `DynamicType` interface. This interface is described in Section 7.5.

Table 23 - Kinds of Language Bindings

Data Representation	***Description***	***Reasons and drawbacks***
Plain Language Binding	Each data type is mapped into the most natural "native" construct in the programming language of choice. For example a STRUCT type is mapped into a class in Java where each member of the STRUCT appears as a field in the class.	Advantages: • Natural. Well integrated in the programming language. • Very compact notation. • Very efficient Disadvantages • Requires compile-time knowledge of the data type • Changes require recompilation • Support for type evolution and sparse data can be cumbersome

Dynamic Language Binding	All data types are mapped into a single Language "Dynamic Data" construct which contains operations to do introspection and access the data within.	Advantages: • Does not require compile-time knowledge of the data type • Does not require code-generation • Well suited for type evolution and sparse data Disadvantages • No compile-time checking • More cumbersome to use than plain data objects • May be lower performance to use than plain data objects

7.5.1 Plain Language Binding

This mapping reuses the OMG-standard IDL-to-language mappings [C-MAP, C++-MAP, JAVA-MAP]. It extends the most commonly used of these bindings in order to express the extended IDL constructs defined in this specification.

The following steps define this language binding in all supported programming language for a particular type.

1. First, express the type in IDL as specified in Section 7.3.1.
2. Next, transform any of the new IDL constructs defined in this specification to their IDL2 equivalents, if applicable. These transformations are defined below.
3. Then, apply the OMG Standard IDL to Language Mapping to the IDL in step 2.
4. Finally, apply any programming language-specific transformations to the generated code, if applicable. These transformations are defined below.

7.5.1.1 Annotations and Built-in Annotations

IDL annotations, including the built-in annotations, impact the language binding as defined below.

7.5.1.1.1 Enumerated Constant Values

Constants in an enumeration type may be given explicit values, as defined in Section 7.2.2.3.1. This addition to the language impacts the bindings for C, C++, and Java in the following ways.

7.5.1.1.1.1 C

The OMG-standard IDL-to-C language mapping [C-MAP] transforms an IDL enumeration into a series of `#define` directives, each corresponding to one of the constants in the enumeration. The values to which these definitions correspond shall be the actual values of the enumerated constants on which the definitions are based, whether implicitly or explicitly defined.

7.5.1.1.1.2 C++

The OMG-standard IDL-to-C++ language mapping [C++-MAP] transforms an IDL enumeration into a C++ enumeration. The C++ programming language supports custom values for enumerated constants. Therefore, for any enumerated constant in IDL that bears the Value annotation, the corresponding C++ enumerated constant definition shall be followed by an equals sign ('=') and the value of the data member of the annotation.

7.5.1.1.1.3 Java

The OMG-standard IDL-to-Java mapping [JAVA-MAP] uses the pre-Java 5 "type-safe enumeration" design pattern. The value of each IDL enumerated constant is given in a Java integer constant of the following form:

```
public static final int _<label> = <value>;
```

…where *<label>* is the name of the IDL constant and *<value>* is its numeric value. As per *this* specification, that numeric value shall be set according to the explicit or implicit value assigned according to the operative Type Representation.

7.5.1.1.2 BitSet Types

The language binding for bit set types is defined based on the language binding for enumerations, just as the IDL Type Representation is based on that for enumerations.

For each bit set type defining flags FLAG_0 through FLAG_*n*, the language binding shall be as if there was an enumeration definition like the following:

```
@BitBound(<bit_bound_value>)
enum <TypeName>Bits {
   @Value(1 << <flag_value_0>)
   FLAG_0,
   …
   @Value(1 << <flag_value_n>)
   FLAG_n,
};
```

Furthermore, the language binding shall be as if there was a typedef like the following, used to represent collections of flags from the previously defined enumeration:

```
typedef <unsigned_integer_equivalent> <TypeName>;
```

…where the type *<unsigned_integer_equivalent>* is chosen based on the bound of the bit set type as defined in the following table.

Bit Set Bound	*Unsigned Integer Equivalent*
1–16	`unsigned short`
17–32	`unsigned long`
33–64	`unsigned long long`

Figure 25 - Bit set integer equivalents

For example, consider the following IDL definition:

```
@BitSet @BitBound(19)
enum MyFlags {
   FIRST_FLAG,
   @VALUE(14)
   SECOND_FLAG,
   THIRD_FLAG,
};
```

The language binding shall be as if the previous definition were replaced by the following:

```
enum MyFlagsBits {
   @Value(1 << 0)
   FIRST_FLAG,
   @VALUE(1 << 14)
   SECOND_FLAG,
   @VALUE(1 << 15)
   THIRD_FLAG,
};
typedef unsigned long MyFlags;
```

7.5.1.1.3 Shareable Members

The storage for a member of an aggregated type may be declared to be external to the storage of the enclosing object of that type. This concept impacts the language bindings for C, C++, and Java in the following ways.

7.5.1.1.3.1 C

Shareable members shall be represented using pointers. Specifically:

- String and wide string members are already represented using pointers, so the mappings for these members do not change. The same shall apply to aliases to string and wide string types.
- Other shareable members are mapped like non-shareable members except that a member of type *X* shall instead be mapped as type *pointer-to-X*. For example, `short` shall be replaced by `short*`.

7.5.1.1.3.2 C++

Shareable members shall be represented using plain pointers rather than automatic values or smart pointers.

- In cases where the non-shareable mapping already uses a plain pointer, it shall remain unchanged.
- In cases where the non-shareable mapping uses a "`_var`" smart pointer, the `_var` type shall be replaced by the corresponding plain pointer type. For example, `MyType_var` is replaced by `MyType*`.
- In cases where the non-shareable mapping uses an automatic member of type *X*, *X* shall be replaced by *pointer-to-X*. For example, `short` shall be replaced by `short*`.

7.5.1.1.3.3 Java

Shareable members shall be represented using object references. Since *all* objects are referred to by reference in Java, the mappings for shareable members of non-primitive types are identical to those of non-shareable members. For IDL types that map to Java primitive types, those Java primitive types shall be replaced by the corresponding object box types from the `java.lang` package. For example, `short` shall be replaced by `java.lang.Short`.

7.5.1.1.4 Nested Types

An IDL compiler need not (although it may) generate `TypeSupport`, `DataReader`, or `DataWriter` classes for any nested type.

7.5.1.1.5 User-Defined Annotation Types

A type designer may define his or her own annotation types. The language bindings for these shall be as follows in Java. In programming languages that lack the concept of annotations, an implementation of this specification may choose to ignore user-defined annotations with respect to this language binding.

7.5.1.1.5.1 Java

Each user-defined IDL annotation type shall be represented by a corresponding Java annotation type. An IDL annotation type defining operations `op_1` through `op_n` shall be represented by the following Java annotation types:

```
public @interface <TypeName> {
    <op_1_type> <op_1_name>() [default <default>];
    …
    <op_n_type> <op_n_name>() [default <default>];
}

public @interface <TypeName>Group {
    <TypeName>[] value();
}
```

The <*op_type*> shall be the Java type corresponding to the return type of the IDL operation. If a default value is specified for a given member, it shall be reflected in the Java definition. Otherwise, the Java definition shall have no default value.

A Java annotation type may itself be annotated (for example, by annotation types in the `java.lang.annotation` package). The presence or absence of any such annotations is undefined.

For each IDL element to which a single instance user-defined annotation is applied, the corresponding Java element shall be annotated with the Java annotation of the same name. For each IDL element to which multiple instances of the annotation are applied, the corresponding Java element shall be annotated with the generated annotation bearing the "Group" suffix; each application of the user-defined annotation shall correspond to a member of the array in the group.

7.5.1.2 Map Types

The language binding for map types is defined by an equivalent IDL2 with exceptions for the C++ and Java language where there is native type support for this type.

7.5.1.2.1 C++

Following the example of the OMG-standard C++ mapping of IDL modules, this extension to the IDL-to-C++ mapping [C++-MAP] is available in two variants based on differences in C++ tool chain compatibility:

- The C mapping defined above remains legal for C++. This mapping avoids issues with older C++ tool chains that may not support namespaces and/or the Standard Template Library (STL).
- An implementation based on the C++-standard `std::map` template is also legal and is defined below.

The C++ Standard [C++-LANG] defines the map container as follows:

```
namespace std {
   template<class Key,
            class T,
            class Compare   = less<Key>,
            class Allocator = allocator<pair<const Key,T> >
   > class map;
}
```

An IDL map type shall be transformed into an instantiation of the `std::map` template such that the `Key` parameter is the C++ type corresponding to the IDL key element type and the `T` parameter is the C++ type corresponding to the IDL value element type. The instantiations for the `Compare` and `Allocator` parameters are undefined and may or may not take their default values.

7.5.1.2.2 Java

An IDL map type shall be represented in Java by an implementation of the standard `java.util.Map` interface. The implementation class to be used is not defined, nor is it defined whether Java 5+ generic syntax should be used. (The OMG-standard IDL-to-Java mapping [JAVA-MAP] predates Java 5, and implementations of it may retain compatibility with earlier versions of Java.)

The key objects for such maps shall be of the Java type corresponding to the IDL key element type. The value objects shall be of the Java type corresponding to the IDL value element type. If either of these Java types is a primitive type, then the corresponding object box type (e.g., `java.lang.Integer` for `int`) shall be used in its place.

7.5.1.2.3 Other Programming Languages

In all languages for which no language-specific mapping is specified, the language binding for map types shall be based on the equivalent IDL2 definition given in Section 7.4.1.1.4.

7.5.1.3 Structure and Union Types

The Plain Language Binding for structure and union types shall correspond to the IDL-to-programming language mappings for IDL structures and unions as amended below.

7.5.1.3.1 Inheritance

A structure type that inherits from another shall be represented as follows.

7.5.1.3.1.1 C++

The C++ `struct` corresponding to the subtype shall publicly inherit from the C++ `struct` corresponding to the supertype.

7.5.1.3.1.2 Java

The Java class corresponding to the subtype shall extend the Java class corresponding to the supertype.

7.5.1.3.1.3 Other Programming Languages

The language binding shall be generated as if an instance of the base type were the first member of the subtype with the name "parent," as in the following equivalent IDL2 definition:

```
struct <struct_name> {
   <base_type_name> parent;
   // ... other members
};
```

7.5.1.3.2 Optional Members

A member of an aggregated type may be declared to be optional, meaning that its value may be omitted from sample to sample of that type. This concept impacts the language bindings for C, C++, and Java in the following ways.

7.5.1.3.2.1 C

Optional members shall be represented using pointers. Specifically:

- String and wide string members are already represented using pointers, so the mappings for these members shall not change. The same shall apply to aliases to string and wide string types.
- Other optional members are mapped like non-optional members except that a member of type *X* shall instead be mapped as type *pointer-to-X*. For example, `short` shall be replaced by `short*`.

A `NULL` pointer shall indicate an omitted value.

7.5.1.3.2.2 C++

Optional members shall be represented using plain pointers rather than automatic values or smart pointers.

- In cases where the mapping of non-optional members already uses a plain pointer, it shall remain unchanged.
- In cases where the mapping of non-optional members uses a "`_var`" smart pointer, the `_var` type shall be replaced by the corresponding plain pointer type. For example, `MyType_var` is replaced by `MyType*`.
- In cases where the mapping of non-optional members uses an automatic member of type *X*, *X* shall be replaced by *pointer-to-X*. For example, `short` shall be replaced by `short*`.

A `NULL` pointer shall indicate an omitted value.

7.5.1.3.2.3 Java

Optional members shall be represented using object references. Since *all* objects are referred to by reference in Java, the mappings for optional members of non-primitive types are identical to those of non-optional members. For IDL types that map to Java primitive types, those Java primitive types shall be replaced by the corresponding object box types. For example, `short` shall be replaced by `java.lang.Short`.

A `null` pointer shall indicate an omitted value.

7.5.2 Dynamic Language Binding

The Dynamic Type Language Binding provides an API to manipulate types. This includes constructing new types as well as introspecting existing types. The API is the same regardless of the Type, allowing applications to manipulate types that were not known at compile time. This API is similar in purpose to the `java.lang.Class` class in Java.

The Dynamic Data Language Binding provides an API to manipulate objects of any Type. This includes creating data objects, setting fields and getting fields, as well as accessing the Type associated with the data object. The API is the same regardless of the type of the object, allowing applications to manipulate data objects of types not known at compile time.

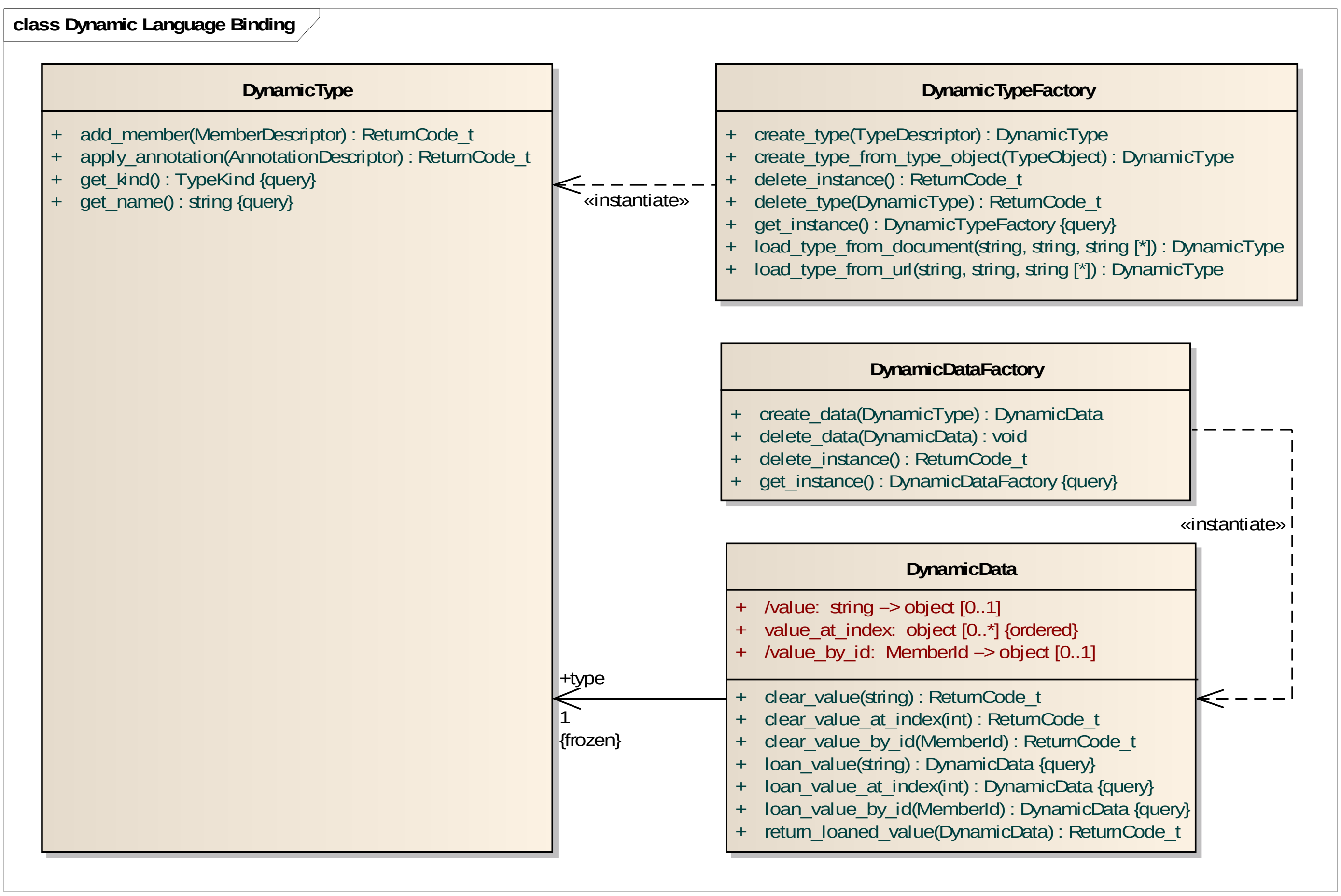

Figure 26 - Dynamic Data and Dynamic Type

There are a small number of fundamental classes to understand in this model, as well as a few helper classes:

- **DynamicType**: Objects of this class represent a type's schema: its physical name, type kind, member definitions (if any), and so on.
- **DynamicTypeFactory**: This type is logically a singleton. Its instance is responsible for creating `DynamicType` and `DynamicTypeSupport` objects.
- **DynamicData**: A `DynamicData` object represents an individual data sample. It provides reflective getters and setters for the members of that sample.
- **DynamicDataFactory**: This type is logically a singleton. Its instance is responsible for creating `DynamicData` objects.

7.5.2.1 UML-to-IDL Mapping Rules

Each type in this Language Binding has an equivalent IDL API. These APIs are specified using the IDL Type Representation defined in this document with the addition of other standard IDL syntax. These latter parts of IDL are used to describe portions of the UML model that have requirements that go beyond those addressed by the IDL Type Representation (for example, local operations).

Specifically, UML constructs shall be mapped to IDL as described below.

- UML enumerations are mapped to IDL enumerations.
- UML classifiers with value semantics are represented as IDL valuetypes. Classifiers with reference semantics are represented as local interfaces.
- UML structural properties in most cases are represented as IDL fields or attributes.
    - Properties of classifiers mapped as valuetypes are represented as plain fields. Properties of classifiers mapped as interfaces are represented as attributes; if the property value is read-only, so is the attribute.
    - Properties with multiplicity [1] (the default if not otherwise noted) are mapped as-is.
    - Properties with multiplicity [0..1] are defined as `@Optional`.
    - Properties with multiplicity [*] (equivalent to [0..*]) or [1..*] may be mapped *either* simply as sequences (in cases where the number of objects is expected to be small and the required level of abstraction low) *or*—in more complex scenarios—a set of methods:

      ```
      unsigned long get_<property_name>_count();
      DDS::ReturnCode_t get_<property_name>(
         inout <property_type> value,
         in unsigned long idx);
      ```

      In addition, if and only if the property value can be modified:

      ```
      DDS::ReturnCode_t set_<property_name>(
         in unsigned long idx,
         in <property_type> value);
      ```

      The "get" operation shall fail with `RETCODE_BAD_PARAMETER` if the given index is outside of the current range. The "set" operation shall do the same with one exception: it shall allow an index one past the end (*i.e.* equal to the current count); setting with this index shall have the effect of appending a new value to the end of the collection. Either operation shall fail with `RETCODE_BAD_PARAMETER` if either argument is nil.

Each type mapping below indicates which of these two mappings it uses in which cases.

- Qualified association ends (representing mappings from one value to another) are mapped to a set of operations:

```
DDS::ReturnCode_t get_<property_name>(
   inout <property_type> value,
   in <qualifier_type> key);
DDS::ReturnCode_t get_all_<property_name>(
   inout map< <qualifier_type>, <property_type> > value);
```

  In addition, if and only if the property value can be modified:

```
DDS::ReturnCode_t set_<property_name>(
   in <qualifier_type> key,
   in <property_type> value);
```

  The “get” operation shall return with `RETCODE_NO_DATA` if no value exists for the given key. Either operation shall return with `RETCODE_BAD_PARAMETER` if either argument is nil.

- UML operations are represented as IDL operations.
  - Static operations are commented, as IDL does not formally support static operations. It is up to the implementer to reflect these operations properly in each programming language to which the IDL may be transformed.

These rules may be qualified or overridden below on a case-by-case basis.

The complete IDL API can be found in “Annex C: Dynamic Language Binding.”

7.5.2.2 DynamicTypeFactory

This class is logically a singleton (although it need not technically be a singleton in practice). Its “only” instance is the starting point for creating and deleting `DynamicType` and `DynamicTypeSupport` objects.

DynamicTypeFactory		
Operations		
static get_instance		DynamicTypeFactory
static delete_instance		ReturnCode_t
get_primitive_type		DynamicType
	kind	TypeKind
create_type		DynamicType
	descriptor	TypeDescriptor
create_type_from_type_object		DynamicType
	type_object	TypeObject

create_string_type		DynamicType
	bound	UInt32
create_wstring_type		DynamicType
	bound	UInt32
create_sequence_type		DynamicType
	element_type	DynamicType
	bound	UInt32
create_array_type		DynamicType
	element_type	DynamicType
	bound	UInt32 [1..*]
create_map_type		DynamicType
	key_element_type	DynamicType
	element_type	DynamicType
	bound	UInt32
create_bitset_type		DynamicType
	bound	UInt32
load_type_from_url		DynamicType
	document_url	string<Char8>
	type_name	string<Char8>
	include_paths	string<Char8> [*]
load_type_from_document		DynamicType
	document	string<Char8>
	type_name	string<Char8>
	include_paths	string<Char8> [*]
delete_type		ReturnCode_t
	type	DynamicType

Figure 27 - `DynamicTypeFactory` properties and operations

7.5.2.2.1 Operation: **`create_array_type`**

Create and return a new `DynamicType` object representing an array type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All array types having equal element types, an equal number of dimensions, and equal bounds in each dimension shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `element_type` - The type of all objects that can be stored in an array of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `bound` - A collection of unsigned integers, the length of which is equal to the number of dimensions in the new array type, and the values of which are the bounds of each dimension. (For example, a three-by-two array would be described by a collection of length two, where the first element had a value of three and the second a value of two.) If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.2.2 Operation: **`create_bitset_type`**

Create and return a new `DynamicType` object representing a bit set type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

If an error occurs, this method shall return a nil value.

Parameter `bound` - The number of reserved bits in the bit set. If this value is out of range, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.2.3 Operation: **`create_map_type`**

Create and return a new `DynamicType` object representing a map type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All map types having equal key and value element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `key_element_type` - The type of all objects that can be stored as keys in a map of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `element_type` - The type of all objects that can be stored as values in a map of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `bound` - The maximum number of key-value pairs that may be stored in a map of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the map type shall be considered to be unbounded.

7.5.2.2.4 Operation: **`create_sequence_type`**

Create and return a new `DynamicType` object representing a sequence type. All objects returned by this operation should eventually be deleted by calling `delete_type`.

All sequence types having equal element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `element_type` - The type of all objects that can be stored in a sequence of the new type. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

Parameter `bound` - The maximum number of elements that may be stored in a map of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the sequence type shall be considered to be unbounded.

7.5.2.2.5 Operations: `create_string_type`, `create_wstring_type`

Create and return a new `DynamicType` object representing a string type. The element type of the result returned by `create_string_type` shall be `Char8`. The element type of the result returned by `create_wstring_type` shall be `Char32`.

All string types having equal element types and equal bounds shall be considered equal. An implementation may therefore elect whether to always return a new object from this method or whether to pool objects and to return previously created type objects consistent with these rules.

If an error occurs, this method shall return a nil value.

Parameter `bound` - The maximum number of elements that may be stored in a string of the new type. If this argument is equal to `LENGTH_UNLIMITED`, the string type shall be considered to be unbounded.

7.5.2.2.6 Operation: `create_type`

Create and return a new `DynamicType` object as described by the given type descriptor. This method is the conventional mechanism for creating structured, enumeration, and alias types, although it can also be used to create types of other kinds. All objects returned by this operation should eventually be deleted by calling `delete_type`.

Parameter `descriptor` - The properties of the new type to create. If this argument is nil or inconsistent (as indicated by its `is_consistent` operation), this operation shall fail and return a nil value.

7.5.2.2.7 Operation: `create_type_from_type_object`

Create and return a new `DynamicType` object that describes a type identical to that described by the given `TypeObject` object. Subsequent changes to the new `DynamicType` object shall not be reflected in the input `TypeObject` object. All objects returned by this operation should eventually be deleted by calling `delete_type`.

Parameter `type_object` - The initial state of the new type to create.

7.5.2.2.8 Operation: `delete_instance`

Reclaim any resources associated with any object(s) previously returned from `get_instance`. Any references to these objects held by previous callers of this operation may become invalid at the discretion of the implementation.

This operation shall fail with RETCODE_ERROR if it fails for any implementation-specific reason.

7.5.2.2.9 Operation: `delete_type`

Delete the given `DynamicType` object, which was previously created by this factory.

Some “deletions” shall always succeed but shall have no observable effect:

- Deletions of nil
- Deletions of objects returned by `get_primitive_type`

Parameter `type` - The type to delete. If this argument is an object that was already deleted, and the implementation is able to detect that fact (which is not required), this operation shall fail with `RETCODE_ALREADY_DELETED`. If an implementation-specific error occurs, this method shall fail with `RETCODE_ERROR`.

7.5.2.2.10 Operation: `get_instance`

Return a `DynamicTypeFactory` instance that behaves like a singleton, although the caller cannot assume pointer equality for the results of multiple calls. The implementation may return the same object every time or different objects at its discretion. However, if it returns different objects, it shall ensure that they behave equivalently with respect to all programming interfaces specified in this document.

Calling this operation is legal even after `delete_instance` has been called. In such a case, the implementation shall recreate or restore the state of the “singleton” as necessary in order to return a valid object to the caller.

If an error occurs, this method shall return a nil value.

7.5.2.2.11 Operation: `get_primitive_type`

Retrieve a `DynamicType` object corresponding to the indicated primitive type kind.

The memory management regime underlying this method is unspecified. Implementations may return references to pre-created objects, they may return new objects with every invocation, or they may take an intermediate approach (for example, lazily creating but then caching objects). Whatever the implementation, the following invariants shall hold:

If an error occurs, this method shall return a nil value.

Parameter `kind` - The kind of the primitive type whose representation is to be returned. If the given kind does not correspond to a primitive type, the operation shall fail and return a nil value.

7.5.2.2.12 Operation: `load_type_from_url`

Create and return a new `DynamicType` object by parsing the type description at the given URL.

Applications shall be able to reclaim resources associated with the type returned by this method by calling `delete_type`, just as if the resultant type was created by one of the `create` methods of this class.

If an error occurs, this method shall return a nil value.

Parameter `document_url` - A URL that indicates a type description document, which shall be parsed to create the `DynamicType` object. Implementations shall minimally support the `file:` URL scheme and may support additional schemes. Implementations shall minimally support the XML Type Description format for loaded documents and may support additional Type Descriptions. (Implementations are recommended to provide a tool or other means of translating among their supported Type Representations.)

Parameter `type_name` - The fully qualified name of the type to be loaded from the document that is the target of the URL. If no type exists of this name in the document (which will trivially be the case if the name is nil or the empty string), the operation shall fail and return a nil result.

Parameter `include_paths` - A collection of URLs to directories to be searched for additional type description documents that may be included, directly or indirectly, by the document that is the target of `document_url`. The directory in which the target of `document_url` resides shall be considered on the inclusion search path implicitly and need not be included in this collection. Implementations shall minimally support the `file:` URL scheme and may support additional schemes.

7.5.2.2.13 Operation: `load_type_from_document`

Create and return a new `DynamicType` object by parsing the type description contained in the given string.

Applications shall be able to reclaim resources associated with the type returned by this method by calling `delete_type`, just as if the resultant type was created by one of the `create` methods of this class.

If an error occurs, this method shall return a nil value.

Parameter `document` - A type description document, which shall be parsed to create the `DynamicType` object. Implementations shall minimally support the XML Type Description format for loaded documents and may support additional Type Descriptions. (Implementations are recommended to provide a tool or other means of translating among their supported Type Representations.)

Parameter `type_name` - The fully qualified name of the type to be loaded from the `document`. If no type exists of this name in the document (which will trivially be the case if the name is nil or the empty string), the operation shall fail and return a nil result.

Parameter `include_paths` - A collection of URLs to directories to be searched for additional type description documents that may be included, directly or indirectly, by the `document` argument. Implementations shall minimally support the `file:` URL scheme and may support additional schemes.

7.5.2.3 AnnotationDescriptor

An `AnnotationDescriptor` packages together the state of an annotation as it is applied to some element (not an annotation type). `AnnotationDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

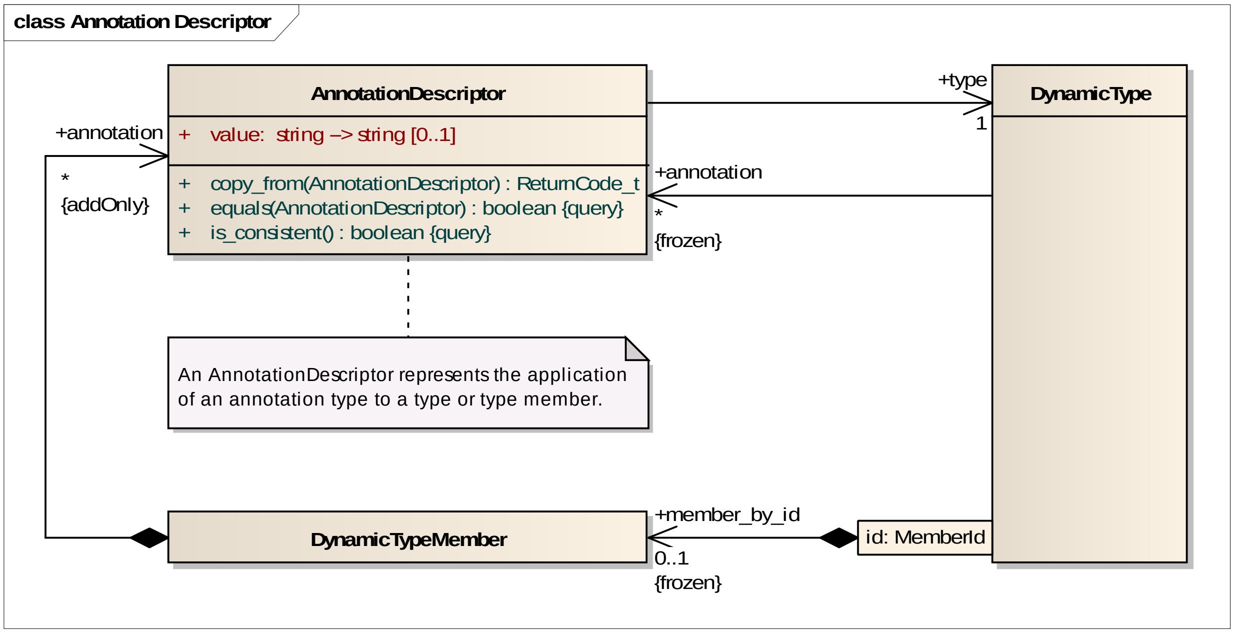

Figure 28 - Annotation Descriptor

AnnotationDescriptor		
Properties		
type	DynamicType	
value	string<Char8,256> → string<Char8,256> [0..1]	
Operations		
copy_from		ReturnCode_t
	other	AnnotationDescriptor
equals		Boolean
	other	AnnotationDescriptor
is_consistent		Boolean

Figure 29 - `AnnotationDescriptor` properties and operations

7.5.2.3.1 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` - The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.3.2 Operation: `equals`

Two annotation descriptors *ad1* and *ad2* are considered equal if and only if all of the following apply:

- Their type properties refer to equal types.
- For every string *s1* for which *ad1*`.value[`*s1*`]` does not exist, *ad2*`.value[`*s1*`]` also does not exist.
- For every string *s1* for which *ad2*`.value[`*s1*`]` does not exist, *ad1*`.value[`*s1*`]` also does not exist.
- For every string *s1* for which *ad1*`.value[`*s1*`]` is a non-nil string *ad1-s2*, *ad2*`.value[`*s1*`]` is a non-nil string *ad2-s2* such that *ad1-s2* equals *ad2-s2*.
- For every string *s1* for which *ad2*`.value[`*s1*`]` is a non-nil string *ad2-s2*, *ad1*`.value[`*s1*`]` is a non-nil string *ad1-s2* such that *ad1-s2* equals *ad2-s2*.

Parameter `other` - Another descriptor to compare to this descriptor. If this argument is nil, this operation shall return `false`.

7.5.2.3.3 Operation: `is_consistent`

Indicate whether this descriptor describes a valid annotation type instantiation. An annotation descriptor is considered consistent if and only if all of the following qualities apply:

- The `type` property refers to a non-nil type of kind `ANNOTATION_TYPE`.
- For every pair of strings *s1* and *s2* such that `value[`*s1*`]` equals `value[`*s2*`]`:
    - String *s1* is the name of an attribute defined by the annotation type referred to by the `type` property.
    - String *s2* is a well-formed string representation of an object of the type of the attribute named by *s1*.

7.5.2.3.4 Property: `type`

The `type` property contains a reference to the annotation type, of which this descriptor describes an instantiation.

When an annotation descriptor is newly created, this reference shall be nil.

7.5.2.3.5 Property: `value`

This property contains a mapping from the names of attributes defined by `type` to valid values of that type. Any attribute defined by `type` but for which no name appears in this property shall be considered to have its default value.

Every attribute value in this property is represented as a string although annotation type members can have other types as well. A string representation of a data value is considered well formed if it would be a valid IDL literal of the corresponding type with the following qualifications:

- String and character literals shall not be surrounded by quotation characters (‘"’ or ‘'’).
- All expressions shall be fully evaluated such that no operators or other non-literal characters occur in the value. For example, “`5`” shall be considered a well-formed string representation of the integer quantity *five*, but “`2 + ENUM_VALUE_THREE`” shall not be.

7.5.2.4 TypeDescriptor

A `TypeDescriptor` packages together the state of a type. `TypeDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

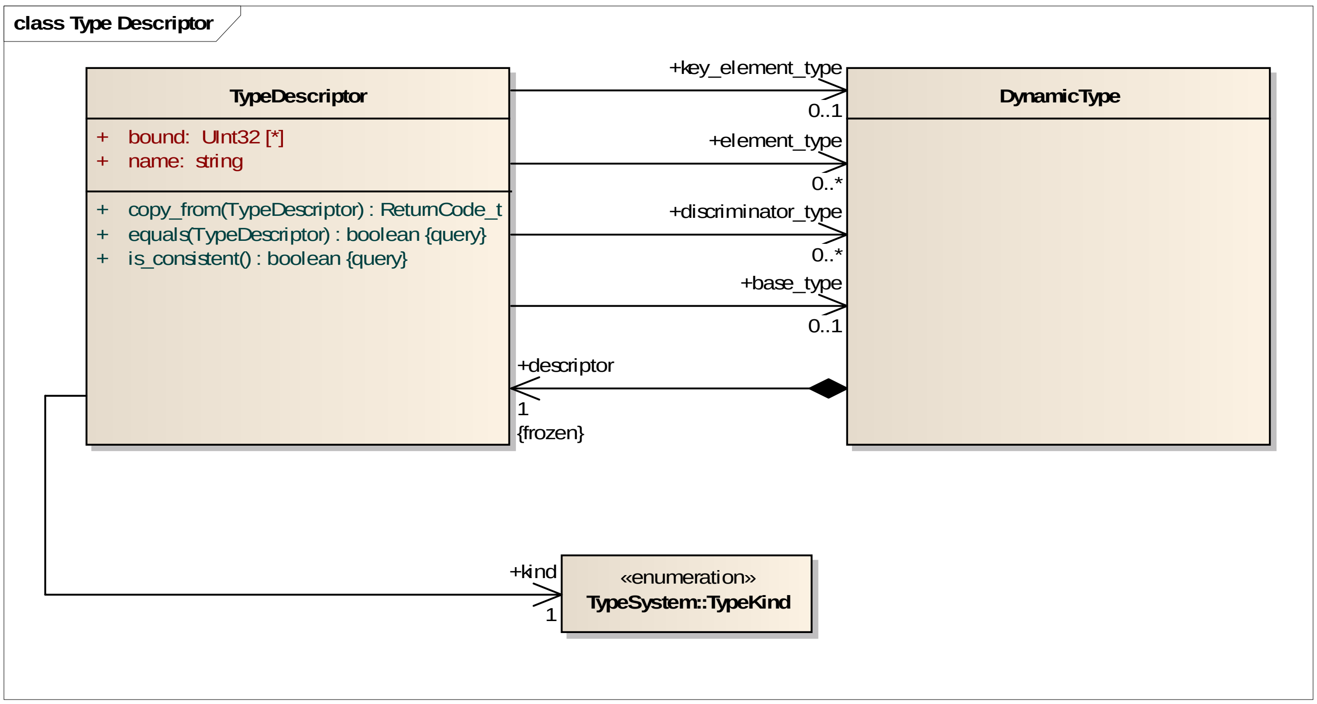

Figure 30 - Type Descriptor

TypeDescriptor		
Properties		
kind	TypeKind	
name	string<Char8,256>	
base_type	DynamicType [0..1]	
discriminator_type	DynamicType [0..1]	
bound	UInt32 [*]	
element_type	DynamicType [0..1]	
key_element_type	DynamicType [0..1]	
Operations		
copy_from		ReturnCode_t
	other	TypeDescriptor
equals		Boolean
	other	TypeDescriptor
is_consistent		Boolean

Figure 31 - `TypeDescriptor` properties and operations

7.5.2.4.1 Property: `base_type`

Another type definition, on which the type described by this descriptor is based. Specifically:

- If this descriptor represents a structure type, `base_type` indicates the supertype of that type. A nil value of this property indicates that the structure type has no supertype.
- If this descriptor represents an alias type, `base_type` indicates the type being aliased. A nil value for this property is not considered consistent.

In all other cases, a consistent descriptor shall have a nil value for this property.

7.5.2.4.2 Property: `bound`

The `bound` property indicates the bound of collection and similar types.

- If this descriptor represents an array type, the length of the property value indicates the number of dimensions in the array, and each value indicates the bound of the corresponding dimension.
- If this descriptor represents a sequence, map, bit set, or string type, the length of the property value is one and the integral value in that property indicates the bound of the collection.

In all other cases, a consistent descriptor shall have a nil value for this property.

7.5.2.4.3 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` - The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.4.4 Property: `discriminator_type`

If this descriptor represents a union type, `discriminator_type` indicates the type of the discriminator of the union. It must not be nil for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for this descriptor to be consistent.

7.5.2.4.5 Property: `element_type`

If this descriptor represents an array, sequence, or string type, this property indicates the element type of the collection. It must not be nil for the descriptor to be consistent.

If this descriptor represents a map type, this property indicates the *value* element type of the map. It must not be nil for the descriptor to be consistent.

If this descriptor represents a bit set type, this property must indicate a Boolean type for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for the descriptor to be consistent.

7.5.2.4.6 Operation: `equals`

Two type descriptors are considered equal if and only if the values of all of the properties identified in the table above are equal in each of them.

Parameter `other` - Another descriptor to compare to this one. If this argument is nil, the operation shall return `false`.

7.5.2.4.7 Operation: `is_consistent`

Indicates whether the states of all of this descriptor's properties are consistent. The definitions of consistency for each property are given in the section corresponding to that property.

7.5.2.4.8 Property: `key_element_type`

If this descriptor represents a map type, this property indicates the *value* element type of the map. It must not be nil for the descriptor to be consistent.

If this descriptor represents any other kind of type, this property must be nil for the descriptor to be consistent.

7.5.2.4.9 Property: `kind`

An enumerated value that indicates what "kind" of type this descriptor describes: a structure, a sequence, etc.

7.5.2.4.10 Property: `name`

The fully qualified name of the type described by this descriptor. To be consistent, this name must be a valid identifier for the given type kind, as defined elsewhere in this document.

7.5.2.5 MemberId

The type `MemberId` is an alias to `UInt32` and is used for the purpose of representing the ID of a member of a structured type.

It is also used to type the constant `MEMBER_ID_INVALID`, which is a sentinel indicating a member ID that is missing, irrelevant, or otherwise invalid in a given context.

7.5.2.6 DynamicTypeMember

A `DynamicTypeMember` represents a "member" of a type. A "member" in this sense may be a member of an aggregated type, a constant within an enumeration, or some other type substructure. Specifically, the behavior is as described in the following figure based on the `TypeKind` of the `DynamicType` to which the member belongs.

Type Kind	***Meaning***
`ANNOTATION_TYPE`	For these aggregated types, a "member" in this sense has the same meaning as it does in the definition of aggregated types generally.
`STRUCTURE_TYPE`	
`UNION_TYPE`	
`BITSET_TYPE`	Each named flag in a bit set shall be considered to be a "member" of that bit set with `Boolean` type.
`ENUMERATION_TYPE`	Each constant in the enumeration shall be considered a "member" of the type. These members shall have the type of the enclosing enumeration itself.
`ALIAS_TYPE`	The behavior is as it would be for the alias's base type.

Figure 32 - `DynamicMember` behavior

No other type kinds are considered to have members.

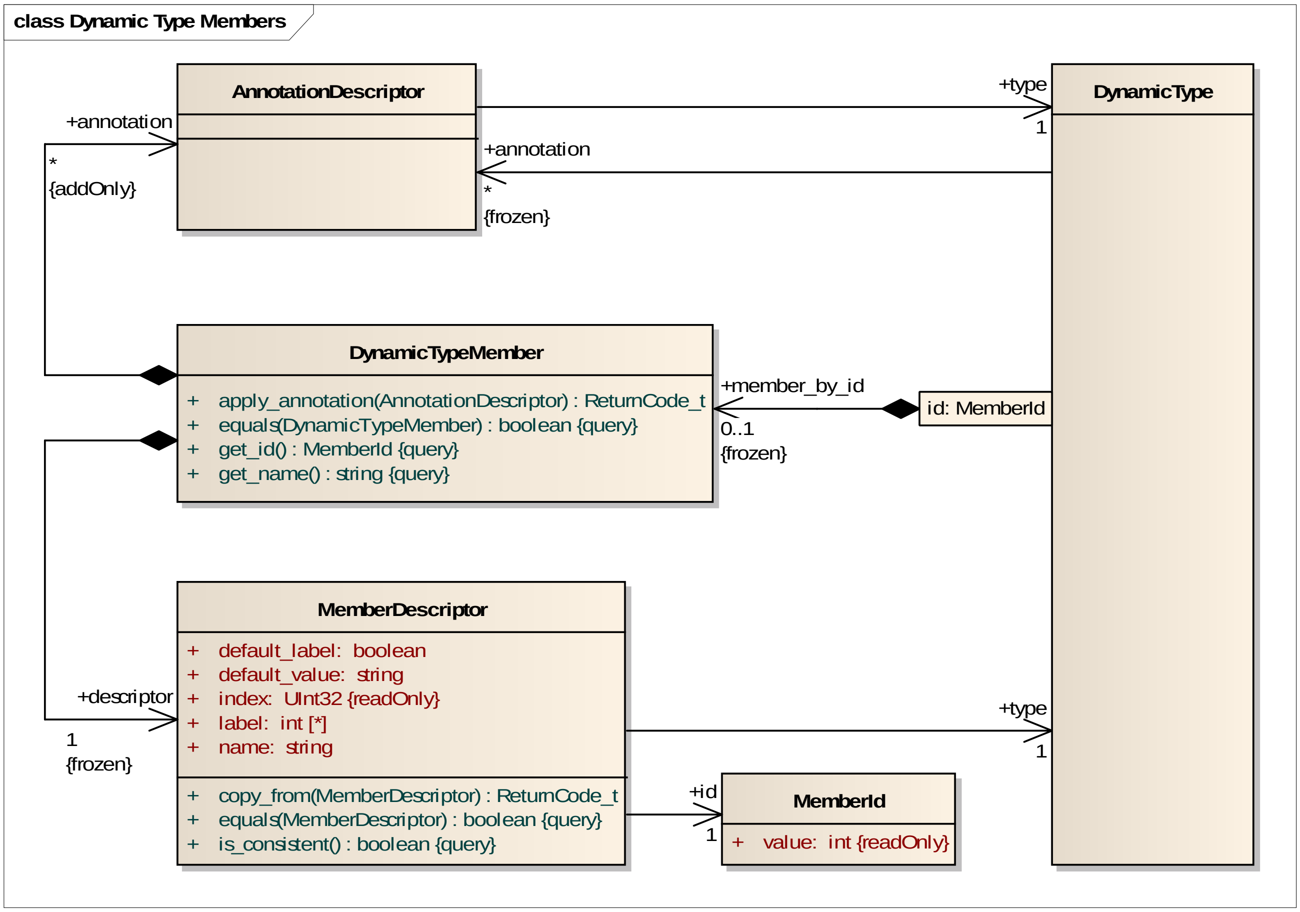

Figure 33 - Dynamic Type Members

`DynamicTypeMember` objects have reference semantics; however, there is an `equals` operation to allow them to be deeply compared.

DynamicTypeMember		
Properties		
descriptor	read-only MemberDescriptor	
annotation	read-only AnnotationDescriptor [*]	
Operations		
equals		Boolean
	other	DynamicTypeMember
apply_annotation		ReturnCode_t
	descriptor	AnnotationDescriptor
get_name		string<Char8,256>
get_id		MemberId

Figure 34 - `DynamicTypeMember` properties and operations

7.5.2.6.1 Property: `annotation`

This property provides all annotations previously applied to this member with `apply_annotation`.

7.5.2.6.2 Operation: `apply_annotation`

Apply the given annotation to this member. It shall subsequently appear in the `annotation` property.

Parameter `descriptor` - A consistent descriptor for the annotation to apply. If this argument is *not* consistent, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.6.3 Property: `descriptor`

This property provides a summary of the state of this member.

7.5.2.6.4 Operation: `equals`

Two members shall be considered equal if and only if they belong to the same type and all of their respective properties, as identified in the table above, are equal.

7.5.2.6.5 Operation: `get_id`

This convenience operation provides the member ID of this member. Its result shall be identical to the ID value that is a member of the `descriptor` property.

7.5.2.6.6 Operation: `get_name`

This convenience operation provides the name of this member. Its result shall be identical to the name string that is a member of the `descriptor` property.

7.5.2.7 MemberDescriptor

A `MemberDescriptor` packages together the state of a `DynamicTypeMember`. `MemberDescriptor` objects have value semantics, allowing them to be deeply copied and compared.

MemberDescriptor		
Properties		
name	String<Char8,256>	
id	MemberId	
type	DynamicType	
default_value	string	
index	read-only UInt32	
label	Int64 [*]	
default_label	Boolean	
Operations		
copy_from		ReturnCode_t
	other	MemberDescriptor
equals		Boolean
	other	MemberDescriptor
is_consistent		Boolean

Figure 35- `MemberDescriptor` properties and operations

7.5.2.7.1 Operation: `copy_from`

Overwrite the contents of this descriptor with those of another descriptor such that subsequent calls to `equals`, passing the same argument as to this method, return `true`. The other descriptor shall not be changed by this operation.

If this operation fails in an implementation-specific way, this operation shall return `RETCODE_ERROR`.

Parameter `other` - The descriptor whose contents are to be copied. If this argument is nil, the operation shall fail with `RETCODE_BAD_PARAMETER`.

7.5.2.7.2 Property: `default_label`

For this descriptor to be consistent, this property must be true if this descriptor identifies the default member of a union type or false if not. A default union member may have additional explicit labels (indicated in the `label` property), but these are semantically irrelevant, as the default member would be in effect or not regardless of their presence or absence.

7.5.2.7.3 Property: `default_value`

This property provides the member's default value in string form. A string representation of a data value is considered well formed if it would be a valid IDL literal of the corresponding type with the following qualifications:

- String and character literals shall not be surrounded by quotation characters ('"' or '").
- All expressions shall be fully evaluated such that no operators or other non-literal characters occur in the value. For example, "`5`" shall be considered a well-formed string representation of the integer quantity *five*, but "`2 + ENUM_VALUE_THREE`" shall not be.

A nil or empty string indicates that the member takes the "default default" value for its type. This rule shall *always* be used when the member is of a type for which IDL provides no syntax to express a literal value (for example, structures or maps) and *may* be used for any other type.

Design rationale: An instance of `DynamicData` might have been used here as an alternative. However, since every default literal can be expressed as a string anyway (i.e., as it is in IDL), and string objects are expected to be more lightweight that `DynamicData` implementations, that representation was preferred.

7.5.2.7.4 Operation: `equals`

Two descriptors are considered equal if and only if the values of all of the properties identified in the table above are equal in each of them.

Parameter `other` - Another descriptor to compare to this one. If this argument is nil, the operation shall return `false`.

7.5.2.7.5 Property: `id`

If this member belongs to an aggregated type, this property indicates the member's ID.

- When a descriptor is used to add a new member to a type, this property may be set to `MEMBER_ID_INVALID`; in that case, the implementation shall select an ID for the new member that is one more than the current maximum member ID in the type. If the value of this property is *not* `MEMBER_ID_INVALID`, it must be set to a value within a legal range.
- When a descriptor is retrieved from an existing member, this property shall reflect the actual ID of the member. It shall therefore not be `MEMBER_ID_INVALID`, and it shall fall within a legal range.

If this member does not belong to an aggregated type, this property *must* be `MEMBER_ID_INVALID`, or the descriptor is not consistent.

7.5.2.7.6 Property: `index`

This property indicates the order of definition of this member within its type, relative to the type's other members. The first member shall have index zero, the next one, and so on.

When a descriptor is used to add a new member to a type, any value greater than the current largest index value in the type shall be taken to indicate that the new member will become the last member, whatever the index; member indices within a type shall not be discontiguous. Alternatively, if this property is set to an index at which a member already exists, that member and all those after it shall be shifted up by a single index value to make room for the new member.

When a descriptor is retrieved from an existing member, this property shall always reflect the actual index at which the member exists.

7.5.2.7.7 Operation: `is_consistent`

A descriptor shall be considered consistent if and only if all of the values of its properties are considered consistent. The meaning of consistency for each of these is defined here in the appropriate section.

7.5.2.7.8 Property: `label`

If the type to which the member belongs is a union, this property indicates the case labels that apply to this member. If `default_label` is false, it must not be empty. In addition, no two members of the same union can specify the same label value.

If the type to which the member belongs is *not* a union, this property's value must be empty to be consistent.

7.5.2.7.9 Property: `name`

This property indicates the name of this member. The value must be a well-formed member name.

7.5.2.7.10 Property: `type`

This property indicates the type of the member's value. It must not be nil, it and must indicate a type that can legally type a member according to the Type System Model.

7.5.2.8 DynamicType

A `DynamicType` object represents a particular type defined according to the Type System. `DynamicType` objects have reference semantics because of the large number of references to them that are expected to exist (e.g., in each `DynamicData` object created from a given `DynamicType`). However, the type nevertheless provides operations to allow copying and comparison by value.

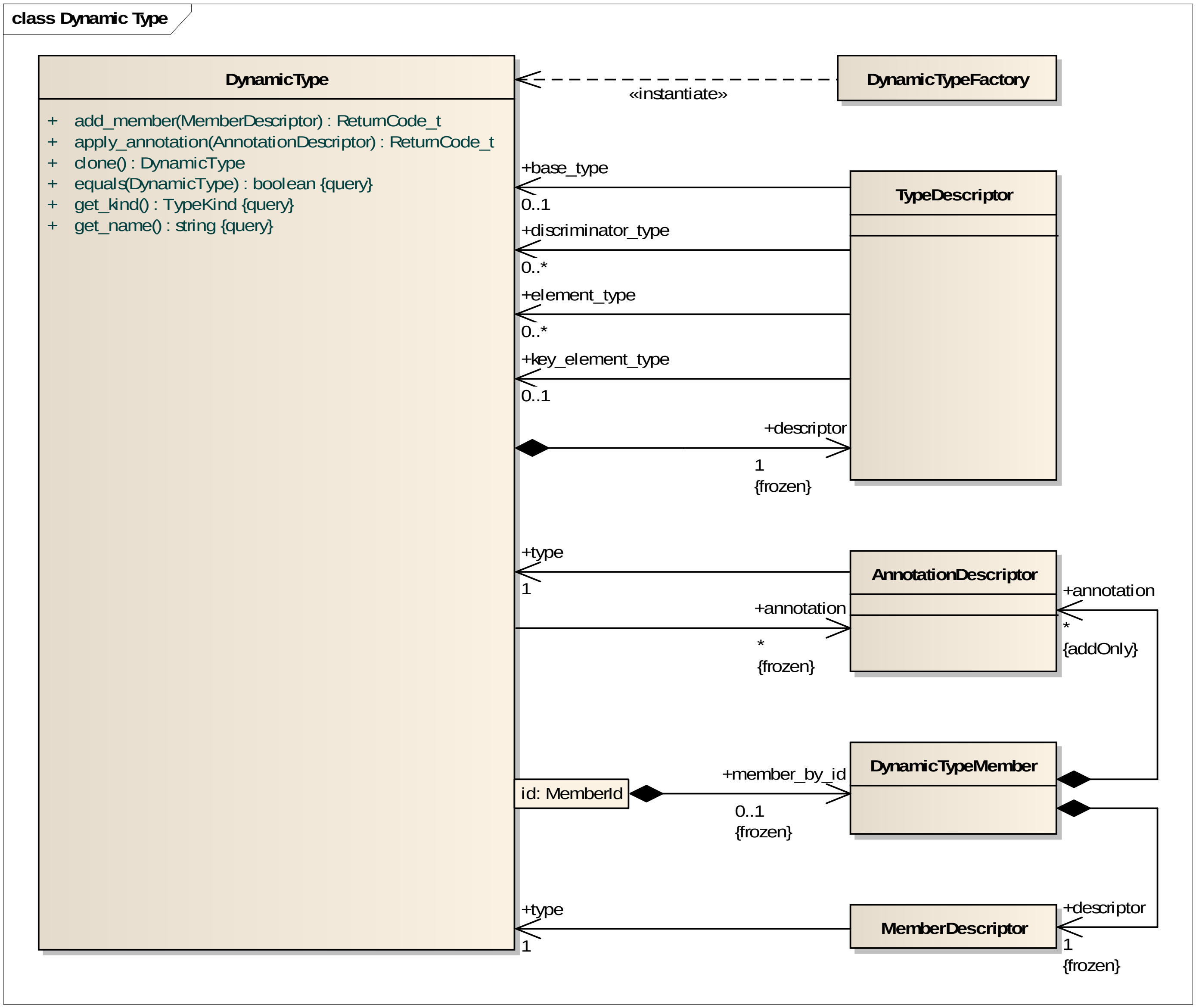

Figure 36 - Dynamic Type

Table 24 - DynamicType properties and operations

DynamicType		
Properties		
descriptor	read-only TypeDescriptor	
member	read-only string<Char8,256> → DynamicTypeMember [0..1]	
member_by_id	read-only MemberId → DynamicTypeMember [0..1]	
annotation	read-only AnnotationDescriptor [*]	
Operations		
equals		Boolean
	other	DynamicType
add_member		ReturnCode_t
	descriptor	MemberDescriptor
apply_annotation		ReturnCode_t
	descriptor	AnnotationDescriptor
clone		DynamicType
get_name		string<Char8,256>
get_kind		TypeKind

7.5.2.8.1 Operation: `add_member`

Add a "member" to this type, where the new "member" has the meaning defined in the specification of the `DynamicTypeMember` class. Specifically, the behavior shall be as described in the following table based on the `TypeKind` of this `DynamicType`.

Table 25 - DynamicType::add_member behavior

Type Kind	*Behavior*
ANNOTATION_TYPE	The member descriptor must describe a consistent annotation type member. If the descriptor does not satisfy these constraints, the operation shall fail with RETCODE_BAD_PARAMETER.
ALIAS_TYPE	The behavior is as it would be for the alias's base type. If adding a member is not defined for the alias's base type, this operation shall fail with RETCODE_PRECONDITION_NOT_MET.
BITSET_TYPE	The member descriptor must describe a Boolean flag with a value within the bound of this bit set type. If the descriptor does not satisfy these constraints, the operation shall fail with RETCODE_BAD_PARAMETER.
ENUMERATION_TYPE	The member descriptor must describe a constant with the type of this enumeration. If the descriptor does not satisfy these constraints, the operation shall fail with RETCODE_BAD_PARAMETER.
STRUCTURE_TYPE	The member descriptor must describe a consistent structure member. If the descriptor does not satisfy this constraint, the operation shall fail with RETCODE_BAD_PARAMETER.
UNION_TYPE	The member descriptor must describe a consistent union member. If the descriptor does not satisfy this constraint, the operation shall fail with RETCODE_BAD_PARAMETER.

For all other type kinds, this operation shall fail with RETCODE_PRECONDITION_NOT_MET.

Following a successful return, the new member shall appear in the member property and possibly in the member_by_id property, based on the definition of that property.

Parameter descriptor - A descriptor of the new member to be added. If this argument is nil, the operation shall fail with RETCODE_BAD_PARAMETER.

7.5.2.8.2 Property: annotation

This property provides all annotations that have previously been applied to this type with apply_annotation.

7.5.2.8.3 Operation: apply_annotation

Apply the given annotation to this type. It shall subsequently appear in the annotation property.

Parameter descriptor - A consistent descriptor for the annotation to apply. If this argument is *not* consistent, the operation shall fail with RETCODE_BAD_PARAMETER.

7.5.2.8.4 Operation: clone

Create and return a new DynamicType object whose state is a copy of this objects state.

7.5.2.8.5 Property: descriptor

This property provides a summary of the state of this type.

7.5.2.8.6 Operation: equals

Two types shall be considered equal if and only if all of their respective properties, as identified in the table above, are equal.

7.5.2.8.7 Operation: get_kind

This convenience operation indicates the kind of this type (e.g., integer, structure, etc.). Its result shall be the same as the kind indicated by the type's descriptor property.

7.5.2.8.8 Operation: get_name

This convenience operation provides the fully qualified name of this type. It shall be identical to the name string that is a member of the descriptor property.

7.5.2.8.9 Property: member

This property contains a mapping from the name of a member of this type to the member itself. As described in the case of add_member, not only members of aggregated types are considered "members" here: the constituents of enumerations, bit sets, and other kinds of types are also considered to be "members" for the purposes of this property.

The lifecycle of a DynamicTypeMember object is governed by that of the DynamicType that contains it. The former shall be considered to exist logically from the time the corresponding member is added to the latter and until such time as the latter is deleted. Implementations may allocate and de-allocate DynamicTypeMember objects more frequently, provided that:

- Users of the DynamicTypeMember class are not required to explicitly delete objects of that class.
- Changes to one DynamicTypeMember object representing a given member shall be reflected in all observable DynamicTypeMember objects representing the same member.
- All DynamicTypeMember objects representing the same member shall compare as equal according to their equals operations.

7.5.2.8.10 Property: member_by_id

This property contains a mapping from the member ID of a member of this (aggregated) type to the member itself.

- If this type is an aggregated type, the collection of members available through this property shall be equal to (element order notwithstanding) that available through the member property.
- If this type is *not* an aggregated type, the collection of members available through this property shall be empty.

7.5.2.9 DynamicDataFactory

This class is logically a singleton (although it need not technically be a singleton in practice). Its "only" instance is the starting point for creating and deleting DynamicData and objects, just like the singleton DomainParticipantFactory is the starting point for creating DomainParticipant objects.

Table 26 - DynamicDataFactory properties and operations

DynamicDataFactory		
Operations		
static get_instance		DynamicDataFactory
static delete_instance		ReturnCode_t
create_data		DynamicData

	type	DynamicType
delete_data		ReturnCode_t
	data	DynamicData

7.5.2.9.1 Operation: `create_data`

Create and return a new data sample. All objects returned by this operation should eventually be deleted by calling `delete_data`.

Parameter `type` - The type of the sample to create.

7.5.2.9.2 Operation: `delete_data`

Dispose of a data sample, reclaiming any associated resources.

Parameter `data` - The data sample to delete.

7.5.2.9.3 Operation: `delete_instance`

Reclaim any resources associated with the object(s) previously returned from `get_instance`. Any references to these objects held by previous callers may become invalid at the implementation's discretion.

This operation shall return `RETCODE_ERROR` if it fails for any implementation-specific reason.

7.5.2.9.4 Operation: `get_instance`

Return a `DynamicDataFactory` instance that behaves like a singleton, although callers cannot assume pointer equality across invocations of this operation. The implementation may return the same object every time or different objects at its discretion. However, if it returns different objects, it shall ensure that they behave equivalently with respect to all programming interfaces specified in this document.

It is legal to call this operation even after `delete_instance` has been called. In such a case, the implementation shall recreate or restore the "singleton" as necessary to ensure that it can return a valid object to the caller.

If an error occurs, this method shall return a nil value.

7.5.2.10 DynamicData

Each object of the `DynamicData` class represents a corresponding object of the type represented by the `DynamicData` object's `DynamicType`.

`DynamicData` objects have reference semantics; however, there is an `equals` operation to allow them to be deeply compared.

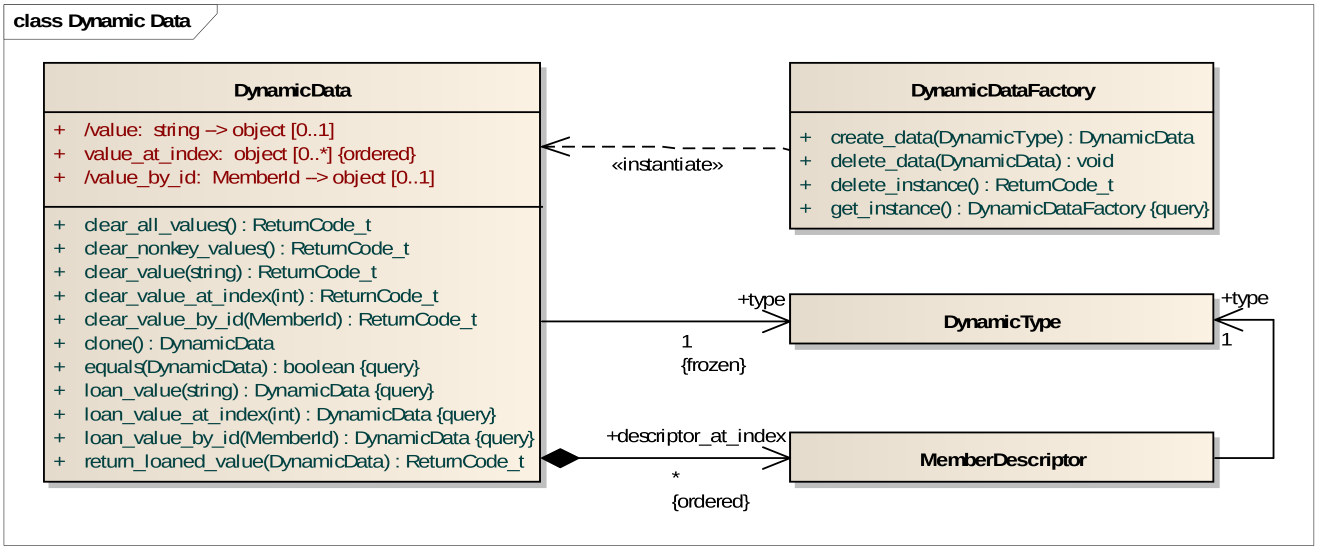

Figure 37 - Dynamic Data and Dynamic Data Factory

The table below summarizes the properties and operations supported by `DynamicData` objects.

Table 27 - DynamicData properties and operations

DynamicData		
Properties		
value	string<Char8,256> → Object [0..1]	
value_by_id	MemberId → Object [0..1]	
value_at_index	Object [*]	
type	read-only DynamicType	
descriptor_at_index	MemberDescriptor [*]	
Operations		
equals		Boolean
	other	DynamicData
clear_all_values		ReturnCode_t
clear_nonkey_values		ReturnCode_t
clear_value		ReturnCode_t
	name	string<Char8,256>
clear_value_by_id		ReturnCode_t
	id	MemberId
clear_value_at_index		ReturnCode_t

	index	UInt32
loan_value		DynamicData
	name	string<Char8,256>
loan_value_by_id		DynamicData
	member_id	MemberId
loan_value_at_index		DynamicData
	index	UInt32
return_loaned_value		ReturnCode_t
	value	DynamicData
clone		DynamicData

7.5.2.10.1 Properties: `value`, `value_at_index`, and `value_by_id`

Many of the properties and operations defined by this class refer to values within the sample, which are identified by name, member ID, or index. What constitutes a value within a sample, and which means of accessing it are valid, depends on the type of this sample.

- If this object is of an aggregated type, values correspond to the type's members and can be accessed by name, member ID, or index.
- If this object is of a sequence or string type, values correspond to the elements of the collection. These elements do not have names or member IDs; they must be accessed by index.
- If this object is of a map type, values correspond to the values of the map. Map keys are implicitly converted to strings and can thus be used to look up map values by name. Map values can also be accessed by index, although the order is unspecified.
- If the object is of an array type, values correspond to the elements of the array. These elements do not have names or member IDs; they must be accessed by index. If the array is multi-dimensional, elements are accessed as if they were "flattened" into a single-dimensional array in the order specified by the IDL specification.
- If the object is of a bit set type, values correspond to the flags within the bit set and are all of `Boolean` type. Named flags can be accessed using that name; any bit within the bound of the bit set may be accessed by its index. These values do not have member IDs.
- If the object is of an enumeration or primitive type, it has no contained values. However, the value of the sample itself may be indicated by "name" using a nil or empty string, by "ID" by passing `MEMBER_ID_INVALID`, or by "index" by passing index 0.

Note that indices used here are always relative to other values *in a particular `DynamicData` object*. Even though member *definitions* within aggregated types have a well-defined order, the same is *not* true within data samples or across data samples. Specifically, the index at which a member of an aggregated type appears in a particular data sample may not match that in which it appears in the corresponding type and may not match the index at which it appears in a different data sample. There are several reasons for these inconsistencies:

- The producer of the sample may be using a slightly different variant of the type than the consumer, which may add to, or omit elements from, the set of members known to the consumer.
- An optional member may have no value; in such a case, it will be omitted, thereby decreasing the index of every subsequent member.
- A non-optional member may likewise be omitted (which semantically is equivalent to it taking its default value). An implementation may discretionarily omit such members (e.g., to save space).
- Preserving member order is not necessary or even desirable (e.g., for performance reasons) for certain data representations.

These three properties provide three different "views" of the values within this data sample: The `value` property provides access to all of those values that can be accessed by name qualified by that name. The `value_by_id` property provides access to all of those values that can be accessed by member ID qualified by that ID. The `value_at_index` property provides access to all values; the sets of values accessible through the other two properties are subsets of the set of values accessible through this property.

7.5.2.10.2 Property: `descriptor_at_index`

This property shall contain a descriptor for each value in this object. The elements in this property shall be parallel to those in the `value_at_index` property.

7.5.2.10.3 Clearing Values: Operations `clear_value`, `clear_value_by_id`, `clear_value_at_index`, `clear_all_values`, and `clear_nonkey_values`

The meaning of "clearing" a member depends on the type of data represented by this sample:

- If this sample is of an aggregated type, and the indicated member is optional, remove it. If the indicated member is not optional, set it to its default value.
- If this sample is of a variable-length collection type, remove the indicated element, shifting any subsequent elements to the next-lowest index.
- If the sample is of an array type, set the indicated element to its default value.
- If the sample is of a bit set type, clear the indicated bit.
- If the sample is of an enumerated type, set it to the first value of the enumerated type.
- If the sample is of a primitive type, set it to its default value.

The `clear_all_members` takes the above action for each value in turn. The `clear_nonkey_value` operation has exactly the same effect as `clear_all_values` with one exception: the values of key fields of aggregated types retain their values.

7.5.2.10.4 Operation: `clone`

Create and return a new data sample with the same contents as this one. A comparison of this object and the clone using `equals` immediately following this call will return `true`.

7.5.2.10.5 Operation: `equals`

Two data samples are considered to be equal if and only if all of the following conditions hold:

- Their respective type definitions are equal.

- All contained values are equal and occur in the same order.
- If the samples' type is an aggregated type, the previous rule shall be amended as follows:
  - Members shall be compared without regard to their order.
  - One of the samples may omit a non-optional member that is present in the other if that member takes its default value in the latter sample.

7.5.2.10.6 Operations: `loan_value`, `loan_value_by_id`, `loan_value_at_index`, and `return_loaned_value`

The "loan" operations loan to the application a `DynamicData` object representing a value within this sample. These operations allow applications to visit values without allocating additional `DynamicData` objects or copying values. This loan shall be returned by the `return_loaned_value` operation.

A given `DynamicData` object may support only a single outstanding loan at a time. That is, after calling a "loan" operation, an application must subsequently call `return_loaned_value` before calling a loan operation again. If an application violates this constraint, the loan operation shall return a nil value.

A loan operation shall also return a nil value if the indicated value does not exist.

The `return_loaned_value` operation shall return `RETCODE_PRECONDITION_NOT_MET` if the provided sample object does not represent an outstanding loan from the sample on which the operation is invoked.

7.5.2.10.7 Property: `type`

This property provides the type that defines the values within this sample. Its value shall not be nil.

7.5.2.10.8 Platform-Specific Model: IDL

The programming language-specific APIs for the Dynamic Type and Dynamic Data classes and their companion classes shall be based on the following IDL definitions, transformed according to the IDL-to-programming language specification above, as expanded below.

The conceptual model refers to the type Object, objects of which may be of any concrete type supported by the Type System defined by this specification. The mapping to IDL below represents this multiplicity of concrete types by multiplying the methods implied by the properties, qualifying each method with a concrete type. For example, a qualified association `foo: Int32 → Object` would expand to `get_int32_foo`, `get_int16_foo`, etc. Specifically, the mapping uses the following type expansions:

- Each primitive type has its own expansion. Primitive types can be implicitly promoted to larger primitive types as defined below.
- Strings of `Char8` and `Char32` elements have their own expansions qualified by "string" and "wstring" respectively.
- Enumerated types shall be implicitly converted to any signed integer type having at least as many bits as the enumerated type's `BitBound`. They are thus accessible through those primitive methods.
- Bit sets shall be implicitly converted to any unsigned integer type having at least as many bits as the bit set's `BitBound`. They are thus accessible through those primitive methods.
- Alias types shall be implicitly converted to their ultimate base type and are thus accessible through the methods appropriate for that type.

- Arrays and sequences of primitive types and strings have their own expansions qualified by the element type's qualifier followed by "array" or "sequence" respectively. Array expansions accept additional arguments—offset and length—to properly support the safe use of sub-arrays in all languages.
- Expansions that operate on `DynamicData` objects, qualified by "complex," catch the remaining cases and offer an alternative approach to accessing values of any of the above types.

As mentioned above, it shall be possible to implicitly promote integral types. These shall be supported during both "get" and "set" operations such that a smaller type promotes to a large type but not visa versa. For example, it shall be possible to get the value of a short integer field as if it were a long integer, and it shall be possible to set the value of a long integer as if it were a short integer. Specifically, the following promotions shall be supported:

- `Int16` → `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Int32` → `Int64`, `Float64`, `Float128`
- `Int64` → `Float128`
- `UInt16` → `Int32`, `Int64`, `UInt32`, `UInt64`, `Float32`, `Float64`, `Float128`
- `UInt32` → `Int64`, `UInt64`, `Float64`, `Float128`
- `UInt64` → `Float128`
- `Float32` → `Float64`, `Float128`
- `Float64` → `Float128`
- `Float128` → (*none*)
- `Char8` → `Char32`, `Int16`, `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Char32` → `Int32`, `Int64`, `Float32`, `Float64`, `Float128`
- `Byte` → (*any*)
- `Boolean` → `Int16`, `Int32`, `Int64`, `UInt16`, `UInt32`, `UInt64`, `Float32`, `Float64`, `Float128`

The complete IDL representation may be found in "Annex C: Dynamic Language Binding."

7.6 Use of the Type System by DDS

This section describes how DDS uses the type system.

7.6.1 Endpoint Matching

The enhanced Type System and the richer set of available Data Representations necessitate extensions to the endpoint matching process defined by the DDS specification, which may be divided into two categories:

- **Data Representation**: The multiplicity of data representations introduced by this specification create the possibility that different `DataWriter` and `DataReader` endpoints in a single system may support different combinations of representations. It is therefore necessary to define a mechanism whereby endpoints can inform each other of the representations they support and thereby negotiate communication.

- **Type Signature**: One of the criteria for DataWriter-DataReader matching defined by DDS is that the type names of each must match exactly. In complex dynamic systems, this restriction can prove overly limiting. Based on the type compatibility rules defined by this specification, matching endpoints shall be permitted to declare types that are not identical but nevertheless have well-defined relationships with one another.

These extensions are defined in the following sections.

7.6.1.1 Data Representation QoS Policy

With multiple standard data Representations available, and vendor-specific extensions possible, DataWriters and DataReaders must be able to negotiate which data representation(s) to use. This negotiation shall occur based on a new QoS policy: DataRepresentationQosPolicy.

7.6.1.1.1 DataRepresentationQosPolicy: Conceptual Model

The conceptual model for data representation negotiation consists of several parts:

- The identification of data representations.
- The specification of supported and preferred representations by DataReaders and DataWriters.
- The algorithm by which a suitable representation is chosen for a particular DataReader/DataWriter pair, given the supported representations of each.

Each data representation shall be identified by a two-byte signed integer value, the "representation identifier." Within the range of such a value, the negative values shall be reserved for definition by DDS implementations. The remainder of the range shall be reserved for the OMG for use in future specifications, including this specification.

Within the OMG-reserved range, this specification defines two representation identifiers: XCDR, which corresponds to the Extended CDR Data Representation and takes the value 0, and XML, which corresponds to the XML Data Representation and takes the value 1.

Each Topic, DataReader and DataWriter shall have a QoS policy DataRepresentationQosPolicy. This policy shall contain a list of representation identifiers. This policy has request-offer semantics [DDS].

- Writers offer a single representation. A writer will use its offered policy to communicate with its matched readers.

  (Because the policy structure includes a sequence, it is technically possible for the writer to offer more than one representation. Implementers of this specification may use this fact in order to offer extended functionality; however, this specification does not specify any meaning for the representation identifiers after the first, and implementations may ignore them.)
- Readers request one or more representations.
- When representations are specified in the TopicQos, the first element of the sequence applies to writers of the Topic, and the whole sequence applies to readers of the Topic.
- If a writer's offered representation is contained within a reader's sequence, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible.

The default value of the DataRepresentationQosPolicy shall be an empty list of preferences. An empty list of preferences shall be taken to be equivalent to a list containing the single element XCDR.

The DataRepresentationQosPolicy shall not be changeable after its corresponding Entity has been enabled.

7.6.1.1.2 Use of the RTPS Encapsulation Identifier

As defined in the RTPS specification, a data encapsulation is identified by a two-byte value, the "encapsulation identifier" [RTPS]. RTPS also identifies specific encapsulation identifier values corresponding to the encapsulations it defines: big-endian CDR, little-endian CDR, big-endian parameter-list CDR, and little-endian parameter-list CDR. These encapsulations correspond to a choice of data representation and a byte-order encoding.

For the purposes of this specification, the two bytes of a representation identifier (an encapsulation identifier) shall be interpreted as a 16-bit unsigned big-endian integral value. Within the range of such a value (from zero [0x0000] to 65,535 [0xFFFF] inclusive), the upper quartile (from 49,152 [0xC000] to 65,535 [0xFFFF] inclusive) shall be reserved for definition by DDS implementations. The remainder of the range shall be reserved for the OMG[10] for use in future specifications, including this specification.

This specification adds an additional encapsulation corresponding to the XML Data Representation: `XML`, with the value 0x0004. (Since XML is a textual format, no byte-order qualification is necessary.)

The encapsulation identifier field in an RTPS data sub-message shall be set such that it corresponds to the data representation of the outermost object whose state is represented in the message. In other words:

- If the Topic is typed by a mutable type, and CDR representation is desired, the RTPS encapsulation identifier shall indicate parameterized CDR encapsulation: `PL_CDR_BE` or `PL_CDR_LE`.
- If the Topic is typed by a final or extensible type, and CDR representation is desired, the RTPS encapsulation identifier shall indicate (plain, compact) CDR encapsulation: `CDR_BE` or `CDR_LE`.
- Regardless of the extensibility kind of the type, if XML representation is desired, the RTPS encapsulation identifier shall by the `XML` identifier defined by this specification.

7.6.1.1.3 `DataRepresentationQosPolicy`: Platform-Specific API

The conceptual model defined above shall be transformed into the IDL definitions `RepresentationId_t`, `RepresentationIdSeq`, `DATA_REPRESENTATION_QOS_POLICY_ID`, `DATA_REPRESENTATION_QOS_POLICY_NAME`, and `DataRepresentationQosPolicy`. These definitions are given in "Annex D: DDS Built-in Topic Data Types."

[10] Note that all RTPS-specified encapsulation identifier values fall within the OMG-reserved range.

7.6.1.2 Type Signature

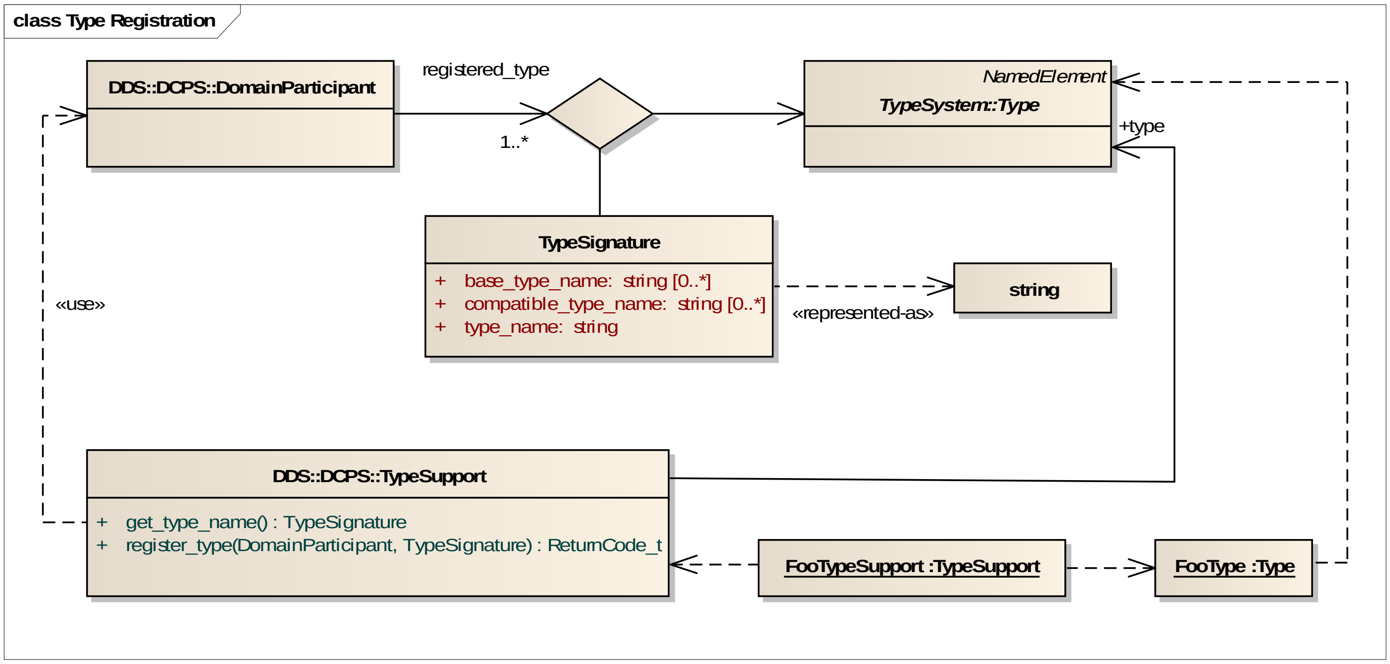

Figure 38 - Type Signature

> *In order for a DDS application to use a type for publication or subscription, the type must first be registered with the corresponding DomainParticipant. When the type is registered, it is given a TypeSignature. The TypeSignature signifies the intended use of the type and is composed of a type_name, a list of equivalent_type_names, and a list of base_type_names. The TypeSignature is used by DDS to determine whether a DataWriter and a DataReader can communicate.*

DDS requires data types to be registered with the DDS `DomainParticipant` prior to using them in the creation of DDS `Topics`. Other than the Built-in Data Types defined in Section 7.6.3 of this specification, all other types must be explicitly registered by invoking the `register_type` operation in the corresponding `TypeSupport` object.

Prior to this (*Extensible and Dynamic Topic Types for DDS*) specification, the type-registration process allowed only the name of the type to be configured at the time the type was registered with the `DomainParticipant`. Other aspects of type were determined from its IDL definition. Moreover, prior to this specification, when a DDS `DataWriter` and a DDS `DataReader` discovered each other, and found out that they both were attached to `Topics` with the same name, they would only check the registered type names of the `Topics` to determine compatibility. If the registered type names matched exactly, then the `DataWriter` was allowed to match the `DataReader`, assuming that the QoS policies were also compatible.

This specification overcomes two limitations of the aforementioned approach. First, it supports the evolution of types—that is, the situation in which the types associated with the `DataWriter` and `DataReader Topics` might not be declared as identical, but might nevertheless be compatible. Second, it mandates strong type checking of the actual structural definition of the type to prevent misconfiguration or programming errors from compromising the integrity of the type system.

To accomplish these goals, this specification modifies the aforementioned behavior in a backwards-compatible manner. The `TypeSupport register_type` operation still takes a string parameter, but this string is interpreted as a `TypeSignature` instead of being interpreted as just a type name. The `TypeSignature` contains three things:

- The name of the type
- The names of other equivalent types, if any—that is, those that may be written and read in either direction.
- The names of other “base” types, if any—that is, those that may be used to read the type but not write it.

These names are used to improve the type-consistency-enforcement algorithm and to allow types to be evolved and to extend each other in a hierarchy.

The equivalent type names and base type names may be expressed as literal strings or as wildcard patterns. The language for these patterns shall be that of the POSIX `fnmatch` function [FNMATCH],

For backwards compatibility with the DDS specification, which uses a single string as a type name, the `TypeSignature` is also represented as a single string. The type signature string incorporates the name of the type being registered, the names of any equivalent types, if any, and the names of any intended base types, if any.

The EBNF grammar for the `TypeSignature` is:

<type_signature>	**::= <type_name> [“,” <equivalent_types>] [“ : ” <base_types>]**
<equivalent_types>	**::= <type_name_pattern> { “,” <type_name_pattern> } ***
<base_types>	**::= <type_name_pattern> { “,” <type_name_pattern> } ***
<type_name>	**::= <typename_literal_char>+**
<type_name_pattern>	**::= <typename_pattern_char>+**
<typename_literal_char>	**::= [a-z] | [A-Z]**
	| [0-9]
	| “_” | “/” | “\” | “.” | “:”
<typename_pattern_char>	**::= <typename_literal_char>**
	| “?” | “*” | “!” | “^” | “[“ | “]”

Note the whitespace character that must surround the “:” in the production rule for **<type_signature>**. This is required to distinguish it from “:” that may appear as a trailing or leading parts of type names. Additional white space surrounding the separator characters “:” and “,” is allowed and shall be ignored.

Note also that the period (‘.’) character has a special meaning within a pattern string. Therefore, to express a literal period character, it must be escaped with a backslash (‘\’). For the sake of consistency, the backslash character may appear in the literal type name as well. Within a literal type name, a double backslash (“\\”) shall be treated as a single literal backslash character; any single occurrence of this character shall be ignored.

The table below lists examples of valid type signatures:

Table 28 - Examples of the string representation of the Type Signature

Type Signature	*Meaning*
Foo	A type named "Foo" with no other equivalent types or types that will be considered base types.
Foo, Foo2, Foo3	A type named "Foo" with which types registered under the names "Foo2" or "Foo3" will be considered equivalent.
Foo : FooBase	A type named "Foo" for which types registered under the name "FooBase" will be considered a base type.
Foo, Foo2, Foo3 : FooBase1, FooBase2	A type named "Foo" with which types registered under the names "Foo2" and "Foo3" will be considered equivalent and for which types registered under the names "FooBase1" or "FooBase2" will be considered base types.

7.6.1.3 Type Consistency Enforcement QoS Policy

The Type Consistency Enforcement QoS Policy defines the rules for determining whether the type used to publish a given data stream is consistent with that used to subscribe to it. It applies to `Topics`, `DataWriters`, and `DataReaders`.

7.6.1.3.1 `TypeConsistencyEnforcementQosPolicy`: Conceptual Model

This policy defines a *type consistency kind*, which allows applications to select from among a set of predetermined policies. The following consistency kinds are specified:

- **`EXACT_TYPE`**: The types published and subscribed to by the `DataWriter` and `DataReader` must be equal, and the registered names of these types must be equal as well. This kind provides the strictest guarantees of type consistency. It corresponds roughly to the degree of type consistency offered by DDS implementations prior to this specification although with added error checking.
- **`EXACT_NAME`**: The types published and subscribed to by the `DataWriter` and `DataReader` must have the same registered names. However, they're structural definitions will be assumed to be consistent without inspection. This is the degree of type consistency enforcement required by the DDS specification [DDS].
- **`DECLARED`**: The type signatures of the `DataWriter` and the `DataReader` must be compatible but need not be identical. The types themselves, TW (type published by the `DataWriter`) and TR (type subscribed to by the `DataReader`), must respect the relationship TR *is-assignable-from* TW, but they need not be identical.
- **`ASSIGNABLE`**: The type signatures are disregarded, and consistency is determined on the basis of type assignability only. The types TW and TR must respect the relationship TR *is-assignable-from* TW.

Further details of these policies are provided in Section 7.6.1.3.2.

This policy has request-offer semantics [DDS]. The type compatibility kind of a `DataWriter` and a `DataReader` must be the same in order for the two to communicate.

The default type consistency kind for `Topics`, `DataWriters`, and `DataReaders` conforming to this specification shall be `DECLARED`. However, when the Service is introspecting the built-in topic data declaration of a remote `DataWriter` or

DataReader in order to determine whether it can match with a local reader or writer, if it observes that no TypeConsistencyEnforcementQosPolicy value is provided (as would be the case when communicating with a Service implementation not in conformance with this specification), it shall assume a kind of EXACT_TYPE[11]. This behavior is consistent with the type member defaulting rules defined in Section 7.2.2.3.5.5, which state that unspecified values of enumeration types take the first value defined for their type—EXACT_TYPE in this case.

7.6.1.3.2 Rules for Type Consistency Enforcement

Implementations of this specification shall use the type-consistency-enforcement rules defined in this section when matching a DataWriter with a DataReader, each associated with a Topic of the same name. These rules are based on the type consistency kind of these entities.

The type-consistency-enforcement rules consist of two steps. The first step checks whether the application programmers have expressed their intent for the two type declarations to be consistent. If this step succeeds, then the second step checks whether the structure of the DataWriter's type is such that it can be assigned to the DataReader's type.

Step 1. To check whether the application developers/integrators have expressed their intent for the two type declarations to be consistent, the Service shall execute the following algorithm:

- If the type compatibility kind is ASSIGNABLE, the Service shall assume that this step passes without performing it, and shall proceed to Step 2 below.
- If the type compatibility kind is EXACT_TYPE or EXACT_NAME, the Service shall compare the type name associated with the DataWriter's topic with that of the DataReader's topic. If the two type names are identical, Step 1 passes and the check proceeds to Step 2; otherwise it fails.
- If the type compatibility kind is DECLARED, the Service shall compare two lists of type names. If these two lists do not have any names in common, Step 1 fails. Otherwise, if there is at least one name in common, Step 1 succeeds and the matching algorithm proceeds to Step 2.
  - *First List*: The following names identify the types that can be written to:
    - The type name associated with the Topic of the DataWriter
    - The names of all the equivalent types of that type, taking into account the type signatures of both the DataWriter's Topic and the DataReader's
    - The names of all the base types of that type, taking into account the type signatures of both the DataWriter's Topic and the DataReader's
  - *Second List*: The following names identify the type names that can be read from:
    - The type name associated with the Topic of the DataReader
    - The names of all the equivalent types of that type, taking into account the type signatures of both the DataReader's Topic and the DataWriter's

Step 2. The type associated with the DataWriter's Topic "TW" and that associated with the DataReader's Topic "TR" are examined to determine whether they are consistent; specifically:

- If the type consistency kind is EXACT_NAME, the Service shall assume that this step passes without performing it.
- If the type consistency kind is EXACT_TYPE, they must satisfy the relationship TR *equals* TW.

---

[11] **Design rationale (non-normative)**: This behavior is critical to ensure that conformant and non-conformant Service implementations reach the same conclusion regarding whether or not a DataWriter and a given DataReader are using consistent types.

- Otherwise, they must satisfy the relationship TR *is-assignable-from* TW, as defined in Section 7.2.4

If they satisfy the above relationship, then Step 2 succeeds; otherwise it fails. If either type TW or TR is not available, then Step 2 shall be assumed to succeed.

If either Step 1 or Step 2 fails, then the `Topic`s associated with the `DataWriter` and `DataReader` are considered to be inconsistent: the `DataWriter` and `DataReader` shall not communicate with each other, and the Service shall trigger an `INCONSISTENT_TOPIC` status change for both the `DataReader`'s `Topic` and the `DataWriter`'s `Topic`.

If both Step 1 and Step 2 succeed, then the `Topic`s are considered to be consistent, and the matching shall proceed to check other aspects of endpoint matching, such as the compatibility of the QoS, as defined by the DDS specification.

Note that the `DataWriter` and the `DataReader` can each execute the algorithm independently, having access to its own `TypeSignature` and the `TypeRepresentation` as well as that of the other endpoint as communicated via DDS discovery (see Section 7.6.2). Moreover, the algorithm is such that both sides are guaranteed to arrive to the same conclusion. That is, either both succeed or both fail.

The table below provides examples with the corresponding match result (for Step 1).

Table 29 - Examples of type consistency checking

`DataWriter` *Type Signature*	***`DataReader`*** *Type Signature*	*Type Compatibility Kind*	*Step1 Match Result and Reason*
Foo	Bar	`DECLARED`	**Fail**. No common type names appear in the two lists.
Foo	Bar	`ASSIGNABLE`	**Succeed**. No common type names are required.
Foo, Bar	Bar	`DECLARED`	**Succeed**. "Bar" is listed as an equivalent type to "Foo."
Foo, Bar	Bar	`EXACT_TYPE` or `EXACT_NAME`	**Fail**. The type names are not the same.
Foo, Cat	Bar, Cat	`DECLARED`	**Succeed**. "Cat" is listed as an equivalent type in both the `DataWriter` and `DataReader` type signatures. That is, there is a transitive equivalency relationship between "Foo" and "Bar" through "Cat."
Foo : Bar	Bar	`DECLARED`	**Succeed**. "Bar" is listed as a base type in the `DataWriter` type signature.
Foo : Cat	Bar, Cat	`DECLARED`	**Succeed**. "Cat" is listed as a base type in the `DataWriter` type signature and listed as an equivalent type in the `DataReader` Type signature.
Foo	Bar : Foo	`DECLARED`	**Fail**. Base types in the `DataReader` are not part of the list of type names compared with the `DataWriter`'s type name list. Intuitively, an object of a base class (Foo) cannot be assigned to an object of a derived class (Bar).

7.6.1.3.3 `TypeConsistencyEnforcementQosPolicy`: Platform-Specific API

The conceptual model defined above shall be transformed into the IDL definitions `TypeConsistencyKind`, `TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_ID`, `TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_NAME`, and `TypeConsistencyEnforcementQosPolicy`. These definitions are given in “Annex D: DDS Built-in Topic Data Types.”

7.6.2 Discovery

This specification enables the description of the DDS built-in topic data types, which are used by the Simple Discovery Protocol of RTPS [RTPS], to be fully described just as would any user-defined types with similar requirements. These expanded (but backwards compatible) definitions may be found in “Annex D: DDS Built-in Topic Data Types.”

The built-in topic data structures shall each contain several additional members:

7.6.2.1 Types Used by Publications and Subscriptions

The topic, publication, and subscription built-in topic data structures shall each indicate the type used for communication by the associated entity as well as the names of the types declared in the type signature to be equivalent or base types. These declarations shall be as follows:

```
@ID(0x0075) @Optional DDS::StringSeq equivalent_type_name;

@ID(0x0076) @Optional DDS::StringSeq base_type_name;

@ID(0x0072) @Optional DDS::TypeObject type;
```

The `the_type` member of the `TypeObject` object shall indicate the type being published and/or subscribed.

An application that wishes to use locally a type it has discovered by means of one of these topics may create a `DynamicTypeSupport` from that `DynamicType` object and register it as usual.

7.6.2.2 QoS Policies

The topic, publication, and subscription built-in topic data types shall each indicate the data representation and type consistency requirements of the associated entity.

```
@ID(0x0073) DDS::DataRepresentationQosPolicy representation;

@ID(0x0074) DDS::TypeConsistencyEnforcementQosPolicy type_compatibility;
```

7.6.3 Built-in Types

DDS shall provide a few types preregistered “out of the box” to allow users to address certain simple use cases without the need for code generation, dynamic type definition, or type registration. These types are:

- **DDS::String**: A single unbounded string; a data type without a key.
- **DDS::KeyedString**: A pair of unbounded strings, one representing the payload and a second representing its key.
- **DDS::Bytes**: An unbounded sequence of bytes, useful for transmitting opaque or application-serialized data.
- **DDS::KeyedBytes**: A payload consisting of an unbounded sequence of bytes plus a key field, an unbounded string.

The built-in types shall be defined as in the following sections and shall be automatically registered by the Service under their fully qualified physical names (as above) with each `DomainParticipant` at the time it is enabled.

Like all non-nested types used with DDS, the built-in types shall have corresponding type-specific `DataWriter` and `DataReader` classes. These shall instantiate the type-specific operations defined by the DDS specification as defined in the following sections; they shall also provide additional overloads.

The built-in types are described briefly below; their complete definitions may be found in “Annex E: Built-in Types.”

7.6.3.1 String

The `DDS::String` type is a simple structure wrapper around a single unbounded string. The wrapper structure exists in order to provide the Service implementation with a non-nested type definition and as a basis of the `TypeObject` object propagated with the built-in topics. But the `StringDataWriter` and `StringDataReader` APIs are defined based on the built-in `string` type for convenience.

7.6.3.2 KeyedString

The `DDS::KeyedString` type is similar to `DDS::String`, but it is a keyed type; the key is an additional unbounded string. `DDS::KeyedStringDataWriter` provides additional overloads that “unwrap” this structure, allowing applications to pass the two strings directly.

7.6.3.3 Bytes

The `DDS::Bytes` type is a simple structure wrapper around a single unbounded sequence of bytes. The wrapper structure exists in order to provide the Service implementation with a non-nested type definition and as a basis of the `TypeObject` object propagated with the built-in topics. The `BytesDataWriter` API is defined based on the underlying sequence for convenience; the `BytesDataReader` API is based on `DDS::Bytes` because of the awkwardness of sequences of sequences.

7.6.3.4 KeyedBytes

The `DDS::KeyedBytes` type is similar to `DDS::Bytes`, but it is a keyed type; the key is an unbounded string. `DDS::KeyedBytesDataWriter` provides additional overloads that “unwrap” this structure, allowing applications to pass the string and sequence directly.

7.6.4 Use of Dynamic Data and Dynamic Type

Using the `DynamicData` and `DynamicType` APIs applications can publish and subscribe data of any type without having compile-type knowledge of the type.

The API is still strongly typed; each specific Type must be registered with the `DomainParticipant`. The `DynamicType` interface can be used to construct the Type and register it with the `DomainParticipant`. The `DynamicData` interface can be used to create objects of a specified Type (expressed by means of a `DynamicType`) and publish and subscribe data objects of that type.

In order to for an application to use a type for publication or subscription the type must first be registered with the corresponding `DomainParticipant` in the same manner as a type defined at compile time.

7.6.4.1 DynamicTypeSupport

The `DynamicTypeSupport` interface extends the `FooTypeSupport` interface defined by the DDS specification where “Foo” is the type `DynamicData`. In addition, it provides access to the type from which it was created.

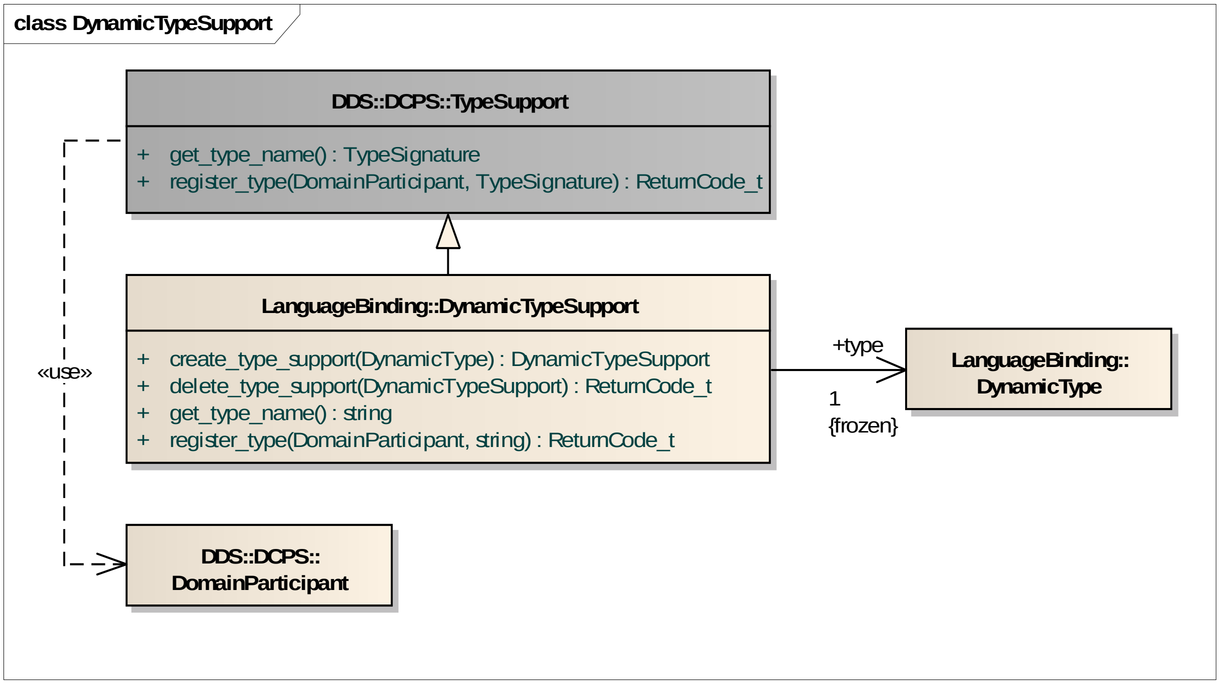

Figure 39 - Dynamic Type Support

DynamicTypeSupport		
Properties		
type	read-only DynamicType	
Operations		
register_type		ReturnCode_t
	participant	DomainParticipant
	type_signature	string<Char8,256>
get_type_name		string<Char8,256>
static create_type_support		DynamicTypeSupport
	type	DynamicType
static delete_type_support		ReturnCode_t
	support	DynamicTypeSupport

Figure 40 - `DynamicTypeSupport` properties and operations

7.6.4.1.1 Property: `type`

Provide a reference to the type definition wrapped by this `DynamicTypeSupport`.

The effect of modifying the type after a `DynamicTypeSupport` has been created is undefined, and the implementation may disallow or ignore such modifications.

7.6.4.1.2 Operations: `register_type`, `get_type_name`

These operations are defined by, and described in, the DDS specification.

7.6.4.1.3 Operation: `create_type_support`

Create and return a new `DynamicTypeSupport` object capable of registering the given type with DDS `DomainParticipants`. The implementation shall ensure that the new type support has a "copy" of the given type object, such that subsequent changes to, or deletions of, the argument object do not impact the new type support. All objects returned by this operation should eventually be deleted by calling `delete_type_support`.

If an error occurs, this method shall return a nil value.

Parameter `type` - The type for which to create a type support. If this argument is nil or is a nested type, the operation shall fail and return a nil value.

7.6.4.1.4 Operation: `delete_type_support`

Delete the given type support object, which was previously created by this factory.

If this argument is nil, the operation shall return successfully without having any observable effect.

Parameter `type_support` - The type support object to delete. If this argument is an object that was already deleted, and the implementation is able to detect that fact (which is not required), this operation shall fail with `RETCODE_ALREADY_DELETED`. If an implementation-specific error occurs, this method shall fail with `RETCODE_ERROR`.

7.6.4.2 `DynamicDataWriter` and `DynamicDataReader`

The `DynamicDataWriter` interface instantiates the `FooDataWriter` interface defined by the DDS specification where "Foo" is the type `DynamicData`.

The `DynamicDataReader` interface instantiates the `FooDataReader` interface defined by the DDS specification where "Foo" is the type `DynamicData`.

These types do not define additional properties or operations.

8. Changes or Extensions Required to Adopted OMG Specifications

8.1 Extensions

8.1.1 DDS

This specification extends the DDS specification [DDS] as described in section 2.1, "*Programming Interface Conformance*," above. As described in that section, these extensions comprise a new, optional conformance level within the DDS specification.

This specification *does not* modify or invalidate any pre-existing DDS profiles or conformance levels, including the Minimum Profile. Therefore, previously conformant DDS implementations remain conformant, and conformance to this additional specification by DDS implementations is completely optional.

8.1.2 IDL

This specification defines several extensions to IDL [IDL] (for example, to represent keys and other DDS-specific features, the syntax of which was previously unspecified). It requires conformance to these extensions only of its own implementations; it does not modify any pre-existing CORBA conformance levels.

8.2 Changes

This specification does not change any pre-existing programming interface, behavior, or other facility of any adopted OMG specification.

Annex A: XML Type Representation Schema

The following XML Schema Document (XSD) formally defines the structure of XML documents conforming to the XML Type Representation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <!-- ================================================================== -->
  <!-- Identifiers                                                          -->
  <!-- ================================================================== -->

  <xs:simpleType name="identifierName">
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-zA-Z]|::)([a-zA-Z_0-9]|::)*"/>
    </xs:restriction>
  </xs:simpleType>


  <!-- ================================================================== -->
  <!-- File Inclusion                                                       -->
  <!-- ================================================================== -->

  <xs:simpleType name="fileName">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>


  <xs:complexType name="includeDecl">
    <xs:attribute name="file"
                  type="fileName"
                  use="required"/>
  </xs:complexType>
```

```
<!-- ================================================================== -->
<!-- Forward Declarations                                               -->
<!-- ================================================================== -->

<xs:simpleType name="forwardDeclTypeKind">
  <xs:restriction base="xs:string">
    <xs:enumeration value="enum"/>
    <xs:enumeration value="struct"/>
    <xs:enumeration value="union"/>
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="forwardDecl">
  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="kind"
                type="forwardDeclTypeKind"
                use="required"/>
</xs:complexType>



<!-- ================================================================== -->
<!-- Basic Types                                                        -->
<!-- ================================================================== -->

<xs:simpleType name="allTypeKind">
  <xs:restriction base="xs:string">
    <!-- Primitive Types -->
    <xs:enumeration value="boolean"/>
    <xs:enumeration value="byte"/>
    <xs:enumeration value="char8"/>
```

```
      <xs:enumeration value="char32"/>
      <xs:enumeration value="int16"/>
      <xs:enumeration value="uint16"/>
      <xs:enumeration value="int32"/>
      <xs:enumeration value="uint32"/>
      <xs:enumeration value="int64"/>
      <xs:enumeration value="uint64"/>
      <xs:enumeration value="float32"/>
      <xs:enumeration value="float64"/>
      <xs:enumeration value="float128"/>

      <!-- String containers -->
      <xs:enumeration value="string"/>
      <xs:enumeration value="wstring"/>

      <!-- Some other type -->
      <xs:enumeration value="nonBasic"/>
    </xs:restriction>
  </xs:simpleType>


  <xs:simpleType name="arrayDimensionsKind">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>


  <!-- ================================================================== -->
  <!-- Constants                                                          -->
  <!-- ================================================================== -->

  <xs:complexType name="constDecl">
    <xs:attribute name="name"
                  type="identifierName"
                  use="required"/>
```

```
  <xs:attribute name="type"
                type="allTypeKind"
                use="required"/>
  <xs:attribute name="nonBasicTypeName"
                type="identifierName"
                use="optional"/>
  <xs:attribute name="value"
                type="xs:string"
                use="required"/>
</xs:complexType>


<!-- =================================================================== -->
<!-- Aggregated Types (General)                                          -->
<!-- =================================================================== -->

<xs:simpleType name="memberId">
  <xs:restriction base="xs:unsignedInt">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="268435455"/><!-- 0x0FFFFFFF -->
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="simpleMemberDecl">
  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="id"
                type="memberId"
                use="optional"/>

  <xs:attribute name="type"
                type="allTypeKind"
                use="required"/>
```

```
    <xs:attribute name="nonBasicTypeName"
                  type="identifierName"
                  use="optional"/>
  </xs:complexType>


  <xs:complexType name="memberDecl">
    <xs:complexContent>
      <xs:extension base="simpleMemberDecl">
        <xs:sequence>
          <xs:element name="annotate"
                      type="annotationDecl"
                      minOccurs="0"
                      maxOccurs="unbounded"/>
        </xs:sequence>

        <xs:attribute name="external"
                      type="xs:boolean"
                      use="optional"
                      default="false"/>
        <xs:attribute name="mustUnderstand"
                      type="xs:boolean"
                      use="optional"
                      default="false"/>

        <xs:attribute name="mapKeyType"
                      type="allTypeKind"
                      use="optional"/>
        <xs:attribute name="mapKeyNonBasicTypeName"
                      type="identifierName"
                      use="optional"/>

        <xs:attribute name="stringMaxLength"
                      type="xs:string"
                      use="optional"/>
```

```
      <xs:attribute name="mapKeyStringMaxLength"
                    type="xs:string"
                    use="optional"/>
      <xs:attribute name="sequenceMaxLength"
                    type="xs:string"
                    use="optional"/>
      <xs:attribute name="mapMaxLength"
                    type="xs:string"
                    use="optional"/>

      <xs:attribute name="arrayDimensions"
                    type="arrayDimensionsKind"
                    use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>


<xs:complexType name="verbatimDecl">
  <xs:sequence>
    <xs:element name="text"
                type="xs:string"
                minOccurs="1"
                maxOccurs="1"/>
  </xs:sequence>

  <xs:attribute name="language"
                type="xs:string"
                use="optional"
                default="*"/>
  <xs:attribute name="placement"
                type="xs:string"
                use="optional"
                default="before-declaration"/>
</xs:complexType>
```

```
<xs:complexType name="structOrUnionTypeDecl">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="annotate"
                  type="annotationDecl"/>
      <xs:element name="verbatim"
                  type="verbatimDecl"/>
    </xs:choice>
  </xs:sequence>

  <xs:simpleType name="extensibilityKind">
    <xs:restriction base="xs:string">
      <xs:enumeration value="final"/>
      <xs:enumeration value="extensible"/>
      <xs:enumeration value="mutable"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="nested"
                type="xs:boolean"
                use="optional"
                default="false"/>
  <xs:attribute name="extensibility"
                type="extensibilitKind"
                use="optional"
                default="false"/>
</xs:complexType>
```

```
<!-- ================================================================== -->
<!-- Annotations                                                         -->
<!-- ================================================================== -->

<xs:complexType name="annotationTypeDecl">
  <xs:sequence>
    <xs:element name="member"
                type="simpleMemberDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="baseType"
                type="identifierName"
                use="optional"/>
</xs:complexType>


<xs:complexType name="annotationMemberValueDecl">
  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="value"
                type="xs:string"
                use="optional"/>
</xs:complexType>


<xs:complexType name="annotationDecl">
  <xs:sequence>
    <xs:element name="member"
                type="annotationMemberValueDecl"
```

```
                minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
</xs:complexType>


<!-- ================================================================== -->
<!-- Structures                                                         -->
<!-- ================================================================== -->

<xs:complexType name="structMemberDecl">
  <xs:complexContent>
    <xs:extension base="memberDecl">
      <xs:attribute name="optional"
                    type="xs:boolean"
                    use="optional"
                    default="false"/>
      <xs:attribute name="key"
                    type="xs:boolean"
                    use="optional"
                    default="false"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>


<xs:complexType name="structDecl">
  <xs:complexContent>
    <xs:extension base="structOrUnionTypeDecl">
      <xs:sequence>
        <xs:choice maxOccurs="unbounded">
```

```
          <xs:element name="member"
                      type="structMemberDecl"
                      minOccurs="1"/>
          <xs:element name="const"
                      type="constDecl"
                      minOccurs="0"/>
        </xs:choice>
      </xs:sequence>

      <xs:attribute name="baseType"
                    type="identifierName"
                    use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>


<!-- ================================================================== -->
<!-- Unions                                                              -->
<!-- ================================================================== -->

<xs:complexType name="unionMemberDecl">
  <xs:complexContent>
    <xs:extension base="memberDecl">
      <!--
      <xs:attribute name="optional"
                    type="xs:boolean"
                    fixed="true"/>
      <xs:attribute name="key"
                    type="xs:boolean"
                    fixed="false"/>
      -->
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="discriminatorDecl">
  <xs:sequence>
    <xs:element name="annotate"
                type="annotationDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="type"
                type="identifierName"
                use="required"/>
  <xs:attribute name="nonBasicTypeName"
                type="identifierName"
                use="optional"/>
  <xs:attribute name="key"
                type="xs:boolean"
                use="optional"
                default="false"/>
  <!--
  <xs:attribute name="optional"
                type="xs:boolean"
                fixed="false"/>
  <xs:attribute name="mustUnderstand"
                type="xs:boolean"
                fixed="true"/>
  -->
</xs:complexType>


<xs:complexType name="caseDiscriminatorDecl">
  <xs:attribute name="value"
                type="xs:string"
                use="required"/>
```

```
</xs:complexType>


<xs:complexType name="caseDecl">
  <xs:sequence>
    <xs:element name="caseDiscriminator"
                type="caseDiscriminatorDecl"
                minOccurs="1"
                maxOccurs="unbounded"/>
    <xs:element name="member"
                type="unionMemberDecl"
                minOccurs="1"
                maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="unionDecl">
  <xs:complexContent>
    <xs:extension base="structOrUnionTypeDecl">
      <xs:sequence>
        <xs:element name="discriminator"
                    type="discriminatorDecl"
                    minOccurs="1"
                    maxOccurs="1"/>
        <xs:element name="case"
                    type="caseDecl"
                    minOccurs="1"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<!-- ==================================================================== -->
<!-- Aliases                                                              -->
<!-- ==================================================================== -->

<xs:complexType name="typedefDecl">
  <xs:attribute name="name"
                type="identifierName"
                use="required"/>

  <xs:attribute name="type"
                type="allTypeKind"
                use="required"/>
  <xs:attribute name="nonBasicTypeName"
                type="identifierName"
                use="optional"/>
  <xs:attribute name="mapKeyType"
                type="allTypeKind"
                use="optional"/>
  <xs:attribute name="mapKeyNonBasicTypeName"
                type="identifierName"
                use="optional"/>

  <xs:attribute name="stringMaxLength"
                type="xs:string"
                use="optional"/>
  <xs:attribute name="mapKeyStringMaxLength"
                type="xs:string"
                use="optional"/>
  <xs:attribute name="sequenceMaxLength"
                type="xs:string"
                use="optional"/>
  <xs:attribute name="mapMaxLength"
                type="xs:string"
                use="optional"/>
```

```
  <xs:attribute name="arrayDimensions"
                type="arrayDimensionsKind"
                use="optional"/>
</xs:complexType>


<!-- ================================================================== -->
<!-- Enumerations                                                       -->
<!-- ================================================================== -->

<xs:simpleType name="enumBitBound">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="32"/>
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="enumeratorDecl">
  <xs:sequence>
    <xs:element name="annotate"
                type="annotationDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="value"
                type="xs:string"
                use="optional"/>
</xs:complexType>
```

```
<xs:complexType name="enumDecl">
  <xs:sequence>
    <xs:element name="annotate"
                type="annotationDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="verbatim"
                type="verbatimDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="enumerator"
                type="enumeratorDecl"
                minOccurs="1"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="bitBound"
                type="enumBitBound"
                use="optional"
                default="32"/>
</xs:complexType>


<!-- ================================================================== -->
<!-- Bit Sets                                                           -->
<!-- ================================================================== -->

<xs:simpleType name="bitsetBitBound">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="64"/>
  </xs:restriction>
```

```
</xs:simpleType>

<xs:simpleType name="flagIndex">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="63"/>
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="flagDecl">
  <xs:sequence>
    <xs:element name="annotate"
                type="annotationDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="value"
                type="flagIndex"
                use="required"/>
</xs:complexType>


<xs:complexType name="bitsetDecl">
  <xs:sequence>
    <xs:element name="annotate"
                type="annotationDecl"
                minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="flag"
                type="flagDecl"
```

```
                minOccurs="0"
                maxOccurs="64"/>
  </xs:sequence>

  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
  <xs:attribute name="bitBound"
                type="bitsetBitBound"
                use="optional"
                default="32"/>
</xs:complexType>


<!-- ================================================================== -->
<!-- Modules                                                            -->
<!-- ================================================================== -->

<xs:group name="moduleElements">
  <xs:sequence>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="include"
                  type="includeDecl"
                  minOccurs="0"/>
      <xs:element name="forward_dcl"
                  type="forwardDecl"
                  minOccurs="0"/>
      <xs:element name="const"
                  type="constDecl"
                  minOccurs="0"/>
      <xs:element name="module"
                  type="moduleDecl"
                  minOccurs="0"/>
      <xs:element name="struct"
                  type="structDecl"
```

```
                  minOccurs="0"/>
      <xs:element name="union"
                  type="unionDecl"
                  minOccurs="0"/>
      <xs:element name="annotation"
                  type="annotationTypeDecl"
                  minOccurs="0"/>
      <xs:element name="typedef"
                  type="typedefDecl"
                  minOccurs="0"/>
      <xs:element name="enum"
                  type="enumDecl"
                  minOccurs="0"/>
      <xs:element name="bitset"
                  type="bitsetDecl"
                  minOccurs="0"/>
    </xs:choice>
  </xs:sequence>
</xs:group>


<xs:complexType name="moduleDecl">
  <xs:group ref="moduleElements"/>
  <xs:attribute name="name"
                type="identifierName"
                use="required"/>
</xs:complexType>


<!-- ============================================================ -->
<!-- Document Root                                                -->
<!-- ============================================================ -->

<xs:element name="types">
  <xs:complexType>
```

```
      <xs:group ref="moduleElements"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Annex B: Representing Types with TypeObject

The following IDL formally describes the `TypeObject` type and those nested types on which it depends.

```
module DDS {
// --- Shared meta-data: -------------------------------------------------

    // All of the kinds of types that exist in the type system
    @BitBound(16)
    enum TypeKind {
        NO_TYPE,    // sentinel indicating "null" value

        BOOLEAN_TYPE,
        BYTE_TYPE,
        INT_16_TYPE,
        UINT_16_TYPE,
        INT_32_TYPE,
        UINT_32_TYPE,
        INT_64_TYPE,
        UINT_64_TYPE,
        FLOAT_32_TYPE,
        FLOAT_64_TYPE,
        FLOAT_128_TYPE,
        CHAR_8_TYPE,
        CHAR_32_TYPE,

        ENUMERATION_TYPE,
        BITSET_TYPE,
        ALIAS_TYPE,

        ARRAY_TYPE,
        SEQUENCE_TYPE,
        STRING_TYPE,
        MAP_TYPE,

        UNION_TYPE,
        STRUCTURE_TYPE,
```

```
    ANNOTATION_TYPE
};

// The name of some element (e.g. type, type member, module)
const long ELEMENT_NAME_MAX_LENGTH = 256;
typedef string<ELEMENT_NAME_MAX_LENGTH> ObjectName;

// Every type has an ID. Those of the primitive types are pre-defined.
typedef long TypeId;

const TypeId NO_TYPE_ID         = NO_TYPE;
const TypeId BOOLEAN_TYPE_ID    = BOOLEAN_TYPE;
const TypeId BYTE_TYPE_ID       = BYTE_TYPE;
const TypeId INT_16_TYPE_ID     = INT_16_TYPE;
const TypeId UINT_16_TYPE_ID    = UINT_16_TYPE;
const TypeId INT_32_TYPE_ID     = INT_32_TYPE;
const TypeId UINT_32_TYPE_ID    = UINT_32_TYPE;
const TypeId INT_64_TYPE_ID     = INT_64_TYPE;
const TypeId UINT_64_TYPE_ID    = UINT_64_TYPE;
const TypeId FLOAT_32_TYPE_ID   = FLOAT_32_TYPE;
const TypeId FLOAT_64_TYPE_ID   = FLOAT_64_TYPE;
const TypeId FLOAT_128_TYPE_ID  = FLOAT_128_TYPE;
const TypeId CHAR_8_TYPE_ID     = CHAR_8_TYPE;
const TypeId CHAR_32_TYPE_ID    = CHAR_32_TYPE;


// --- Annotation usage: -------------------------------------------------

// ID of a type member
typedef unsigned long MemberId;
const MemberId MEMBER_ID_INVALID = 0x0FFFFFFF;
```

```
/* Literal value of an annotation member: either the default value in its
 * definition or the value applied in its usage.
 */
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
union AnnotationMemberValue switch (TypeKind) {
    case BOOLEAN_TYPE:
        boolean boolean_value;
    case BYTE_TYPE:
        octet byte_value;
    case INT_16_TYPE:
        short int_16_value;
    case UINT_16_TYPE:
        unsigned short uint_16_value;
    case INT_32_TYPE:
        long int_32_value;
    case UINT_32_TYPE:
        unsigned long uint_32_value;
    case INT_64_TYPE:
        long long int_64_value;
    case UINT_64_TYPE:
        unsigned long long uint_64_value;
    case FLOAT_32_TYPE:
        float float_32_value;
    case FLOAT_64_TYPE:
        double float_64_value;
    case FLOAT_128_TYPE:
        long double float_128_value;
    case CHAR_8_TYPE:
        char character_value;
    case CHAR_32_TYPE:
        wchar wide_character_value;
    case ENUMERATION_TYPE:
        long enumeration_value;
    case STRING_TYPE:
        wstring string_value;   // use wide str regardless of char width
```

```
};

// The assignment of a value to a member of an annotation
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct AnnotationUsageMember {
    MemberId member;                // member of the annotation type
    AnnotationMemberValue value;    // value that member is set to
};

typedef sequence<AnnotationUsageMember> AnnotationUsageMemberSeq;

// The application of an annotation to some type or type member
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct AnnotationUsage {
    TypeId type;
    AnnotationUsageMemberSeq member;
};

typedef sequence<AnnotationUsage> AnnotationUsageSeq;


// --- Type base class: -----------------------------------------------

// Flags that apply to type definitions
@BitSet @BitBound(16)
enum TypeFlag {
    @Value(0) IS_FINAL,     // | can't both
    @Value(1) IS_MUTABLE,   // | be '1'
    @Value(2) IS_NESTED
};

// Fundamental properties of any type definition
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct TypeProperty {
    TypeFlag flag;
```

```
    TypeId id;
    ObjectName name;
};

// Member IDs used in the Type base type
enum TypeMemberId {
    @Value(0) PROPERTY_TYPE_MEMBER_ID,
    @Value(1) ANNOTATION_TYPE_MEMBER_ID
};

// Base type for all type definitions
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct Type {
    @ID(PROPERTY_TYPE_MEMBER_ID)   TypeProperty property;
    @ID(ANNOTATION_TYPE_MEMBER_ID) AnnotationUsageSeq annotation;
};


// --- Aggregations: -----------------------------------------------------

// Flags that apply to aggregation type members
@BitSet @BitBound(16)
enum MemberFlag {
    @Value(0) IS_KEY,
    @Value(1) IS_OPTIONAL,
    @Value(2) IS_SHAREABLE,
    @Value(3) IS_UNION_DEFAULT  // set if member is union default case
};

// Fundamental properties of any aggregation type member
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct MemberProperty {
    MemberFlag flag;
    MemberId id;
    TypeId type;
```

```
    ObjectName name;
};

// Member IDs used in the Member base type
enum MemberMemberId {
    @Value(0) PROPERTY_MEMBER_MEMBER_ID,
    @Value(1) ANNOTATION_MEMBER_MEMBER_ID
};

// Member of an aggregation type
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct Member {
    @ID(PROPERTY_MEMBER_MEMBER_ID)   MemberProperty property;
    @ID(ANNOTATION_MEMBER_MEMBER_ID) AnnotationUsageSeq annotation;
};

typedef sequence<Member> MemberSeq;

// Member IDs used in the StructureType type
enum StructureTypeMemberId {
    @Value(100) BASE_TYPE_STRUCTURETYPE_MEMBER_ID,
    @Value(101) MEMBER_STRUCTURETYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct StructureType : Type {
    @ID(BASE_TYPE_STRUCTURETYPE_MEMBER_ID) TypeId base_type;
    @ID(MEMBER_STRUCTURETYPE_MEMBER_ID)    MemberSeq member;
};

// Case labels that apply to a member of a union type
typedef sequence<long> UnionCaseLabelSeq;
```

```
// Member IDs used in the UnionMember type
enum UnionMemberMemberId {
    @Value(100) LABEL_UNIONMEMBER_MEMBER_ID
};

// Member of a union type
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct UnionMember : Member {
    @ID(LABEL_UNIONMEMBER_MEMBER_ID) UnionCaseLabelSeq label;
};

typedef sequence<UnionMember> UnionMemberSeq;

// Member IDs used in the UnionType type
enum UnionTypeMemberId {
    @Value(100) MEMBER_UNIONTYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct UnionType : Type {
    @ID(MEMBER_UNIONTYPE_MEMBER_ID) UnionMemberSeq member;
};

// Member IDs used in the AnnotationMember type
enum AnnotationMemberMemberId {
    @Value(100) DEFAULT_VALUE_ANNOTATIONMEMBER_MEMBER_ID
};

// Member of an annotation type
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct AnnotationMember : Member {
    @ID(DEFAULT_VALUE_ANNOTATIONMEMBER_MEMBER_ID)
    AnnotationMemberValue default_value;
};
```

```
typedef sequence<AnnotationMember> AnnotationMemberSeq;

// Member IDs used in the AnnotationType type
enum AnnotationTypeMemberId {
    @Value(100) BASE_TYPE_ANNOTATIONTYPE_MEMBER_ID,
    @Value(101) MEMBER_ANNOTATIONTYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct AnnotationType : Type {
    @ID(BASE_TYPE_ANNOTATIONTYPE_MEMBER_ID) TypeId base_type;
    @ID(MEMBER_ANNOTATIONTYPE_MEMBER_ID)    AnnotationMemberSeq member;
};


// --- Alias: ------------------------------------------------------------

// Member IDs used in the AliasType type
enum AliasTypeMemberId {
    @Value(100) BASE_TYPE_ALIASTYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct AliasType : Type {
    @ID(BASE_TYPE_ALIASTYPE_MEMBER_ID) TypeId base_type;
};


// --- Collections: ------------------------------------------------------

// Bound of a collection type
typedef unsigned long Bound;
typedef sequence<Bound> BoundSeq;
const Bound UNBOUNDED_COLLECTION = 0;
```

```
// Member IDs used in the CollectionType base type
enum CollectionTypeMemberId {
    @Value(100) ELEMENT_TYPE_COLLECTIONTYPE_MEMBER_ID
};

// Base type for collection types
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct CollectionType : Type {
    @ID(ELEMENT_TYPE_COLLECTIONTYPE_MEMBER_ID) TypeId element_type;
};

// Member IDs used in the ArrayType type
enum ArrayTypeMemberId {
    @Value(200) BOUND_ARRAYTYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct ArrayType : CollectionType {
    @ID(BOUND_ARRAYTYPE_MEMBER_ID) BoundSeq bound;
};

// Member IDs used in the MapType type
enum MapTypeMemberId {
    @Value(200) KEY_ELEMENT_TYPE_MAPTYPE_MEMBER_ID,
    @Value(201) BOUND_MAPTYPE_MEMBER_ID
};

@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct MapType : CollectionType {
    @ID(KEY_ELEMENT_TYPE_MAPTYPE_MEMBER_ID) TypeId key_element_type;
    @ID(BOUND_MAPTYPE_MEMBER_ID)            Bound bound;
};
```

```
// Member IDs used in the SequenceType type
enum SequenceTypeMemberId {
    @Value(200) BOUND_SEQUENCETYPE_MEMBER_ID
};

@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct SequenceType : CollectionType {
    @ID(BOUND_SEQUENCETYPE_MEMBER_ID) Bound bound;
};

// Member IDs used in the StringType type
enum StringTypeMemberId {
    @Value(200) BOUND_STRINGTYPE_MEMBER_ID
};

@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct StringType : CollectionType {
    @ID(BOUND_STRINGTYPE_MEMBER_ID) Bound bound;
};


// --- Bit set: ---------------------------------------------------------

// Bit in a bit set
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct Bit {
    long index;
    ObjectName name;
};

typedef sequence<Bit> BitSeq;

// Member IDs used in the BitSetType type
enum BitSetTypeMemberId {
    @Value(100) BIT_BOUND_BITSETTYPE_MEMBER_ID,
```

```
    @Value(101) BIT_BITSETTYPE_MEMBER_ID
};

@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct BitSetType : Type {
    @ID(BIT_BOUND_BITSETTYPE_MEMBER_ID) Bound bit_bound;
    @ID(BIT_BITSETTYPE_MEMBER_ID)       BitSeq bit;
};


// --- Enumeration: ------------------------------------------------------

// Constant in an enumeration type
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct EnumeratedConstant {
    long value;
    ObjectName name;
};

typedef sequence<EnumeratedConstant> EnumeratedConstantSeq;

// Member IDs used in the EnumerationType type
enum EnumerationTypeMemberId {
    @Value(100) BIT_BOUND_ENUMERATIONTYPE_MEMBER_ID,
    @Value(101) CONSTANT_ENUMERATIONTYPE_MEMBER_ID
};

// Enumeration type
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
struct EnumerationType : Type {
    @ID(BIT_BOUND_ENUMERATIONTYPE_MEMBER_ID)
    Bound bit_bound;
    @ID(CONSTANT_ENUMERATIONTYPE_MEMBER_ID)
    EnumeratedConstantSeq constant;
};
```

```
// --- Module: ------------------------------------------------------------

    struct TypeLibrary;

    @Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
    struct Module {
        ObjectName name;
        @Shared TypeLibrary library;
    };


    // --- Type library: -----------------------------------------------------

    // All of the kinds of definitions that can exist in a type library
    @BitBound(16)
    enum TypeLibraryElementKind {
        @Value(ALIAS_TYPE)       ALIAS_TYPE_ELEMENT,
        @Value(ANNOTATION_TYPE)  ANNOTATION_TYPE_ELEMENT,
        @Value(ARRAY_TYPE)       ARRAY_TYPE_ELEMENT,
        @Value(BITSET_TYPE)      BITSET_TYPE_ELEMENT,
        @Value(ENUMERATION_TYPE) ENUMERATION_TYPE_ELEMENT,
        @Value(MAP_TYPE)         MAP_TYPE_ELEMENT,
        @Value(SEQUENCE_TYPE)    SEQUENCE_TYPE_ELEMENT,
        @Value(STRING_TYPE)      STRING_TYPE_ELEMENT,
        @Value(STRUCTURE_TYPE)   STRUCTURE_TYPE_ELEMENT,
        @Value(UNION_TYPE)       UNION_TYPE_ELEMENT,

        /*auto-assigned value*/  MODULE_ELEMENT
    };
```

```
// Element that can appear in a type library or module: a type or module
@Extensibility(MUTABLE_EXTENSIBILITY) @Nested
union TypeLibraryElement switch (TypeLibraryElementKind) {
    case ALIAS_TYPE_ELEMENT:
        AliasType alias_type;
    case ANNOTATION_TYPE_ELEMENT:
        AnnotationType annotation_type;
    case ARRAY_TYPE_ELEMENT:
        ArrayType array_type;
    case BITSET_TYPE_ELEMENT:
        BitSetType bitset_type;
    case ENUMERATION_TYPE_ELEMENT:
        EnumerationType enumeration_type;
    case MAP_TYPE_ELEMENT:
        MapType map_type;
    case SEQUENCE_TYPE_ELEMENT:
        SequenceType sequence_type;
    case STRING_TYPE_ELEMENT:
        StringType string_type;
    case STRUCTURE_TYPE_ELEMENT:
        StructureType structure_type;
    case UNION_TYPE_ELEMENT:
        UnionType union_type;
    case MODULE_ELEMENT:
        Module module;
};

typedef sequence<TypeLibraryElement> TypeLibraryElementSeq;

@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct TypeLibrary {
    TypeLibraryElementSeq element;
};
```

```
    /* Central type of this Type Representation: identifies a single type
     * within a library.
     */
    @Extensibility(MUTABLE_EXTENSIBILITY)
    struct TypeObject {
        @Shared   TypeLibrary library;
        @Optional TypeId the_type;
    };
};  // end module DDS
```

Annex C: Dynamic Language Binding

The following IDL comprises the API for the Dynamic Language Binding.

```
module DDS {
    local interface DynamicType;
    valuetype TypeDescriptor;

    typedef sequence<string> IncludePathSeq;

    local interface DynamicTypeFactory {
        /*static*/ DynamicTypeFactory get_instance();
        /*static*/ DDS::ReturnCode_t delete_instance();

        DynamicType get_primitive_type(in TypeKind kind);
        DynamicType create_type(in TypeDescriptor descriptor);
        DynamicType create_string_type(in unsigned long bound);
        DynamicType create_wstring_type(in unsigned long bound);
        DynamicType create_sequence_type(
            in DynamicType element_type,
            in unsigned long bound);
        DynamicType create_array_type(
            in DynamicType element_type,
            in BoundSeq bound);
        DynamicType create_map_type(
            in DynamicType key_element_type,
            in DynamicType element_type,
            in unsigned long bound);
        DynamicType create_bitset_type(in unsigned long bound);
        DynamicType load_type_from_url(
            in string document_url,
            in string type_name,
            in IncludePathSeq include_paths);
        DynamicType load_type_from_document(
```

```
        in string document,
        in string type_name,
        in IncludePathSeq include_paths);
    DDS::ReturnCode_t delete_type(in DynamicType type);
};

local interface DynamicTypeSupport {
    /*static*/ DynamicTypeSupport create_type_support(
        in DynamicType type);
    /*static*/ DDS::ReturnCode_t delete_type_support(
        in DynamicTypeSupport type_support);

    DDS::ReturnCode_t register_type(
        in DDS::DomainParticipant participant,
        in ObjectName type_signature);
    ObjectName get_type_name();
};

valuetype AnnotationDescriptor  {
    public DynamicType type;

    DDS::ReturnCode_t get_value(
        inout ObjectName value, in ObjectName key);
    DDS::ReturnCode_t get_all_value(
        inout Parameters value);
    DDS::ReturnCode_t set_value(
        in ObjectName key, in ObjectName value);

    DDS::ReturnCode_t copy_from(in AnnotationDescriptor other);
    boolean equals(in AnnotationDescriptor other);
    boolean is_consistent();
};

valuetype TypeDescriptor {
    public TypeKind kind;
```

```
    public ObjectName name;
    public DynamicType base_type;
    public DynamicType discriminator_type;
    public BoundSeq bound;
    @Optional public DynamicType element_type;
    @Optional public DynamicType key_element_type;

    DDS::ReturnCode_t copy_from(in TypeDescriptor other);
    boolean equals(in TypeDescriptor other);
    boolean is_consistent();
};

valuetype MemberDescriptor  {
    public ObjectName name;
    public MemberId id;
    public DynamicType type;
    public string default_value;
    public unsigned long index;
    public UnionCaseLabelSeq label;
    public boolean default_label;

    DDS::ReturnCode_t copy_from(in MemberDescriptor descriptor);
    boolean equals(in MemberDescriptor descriptor);
    boolean is_consistent();
};

local interface DynamicTypeMember {
    readonly attribute MemberDescriptor descriptor;

    unsigned long get_annotation_count();
    DDS::ReturnCode_t get_annotation(
        inout AnnotationDescriptor descriptor,
        in unsigned long idx);

    boolean equals(in DynamicTypeMember other);
```

```
    DDS::ReturnCode_t apply_annotation(
        in AnnotationDescriptor descriptor);

    MemberId get_id();
    ObjectName get_name();
};

typedef map<ObjectName, DynamicTypeMember> DynamicTypeMembersByName;
typedef map<MemberId, DynamicTypeMember>   DynamicTypeMembersById;

local interface DynamicType {
    readonly attribute TypeDescriptor descriptor;

    ObjectName get_name();
    TypeKind get_kind();

    DDS::ReturnCode_t get_member(
        inout DynamicTypeMember member,
        in ObjectName name);
    DDS::ReturnCode_t get_all_members(
        inout DynamicTypeMembersByName member);

    DDS::ReturnCode_t get_member_by_id(
        inout DynamicTypeMember member,
        in MemberId id);
    DDS::ReturnCode_t get_all_members_by_id(
        inout DynamicTypeMembersById member);

    unsigned long get_annotation_count();
    DDS::ReturnCode_t get_annotation(
        inout AnnotationDescriptor descriptor,
        in unsigned long idx);

    boolean equals(in DynamicType other);
    DDS::ReturnCode_t add_member(in MemberDescriptor descriptor);
```

```
    DDS::ReturnCode_t apply_annotation(
        in AnnotationDescriptor descriptor);
    DynamicType clone();
};

local interface DynamicData;

local interface DynamicDataFactory  {
    /*static*/ DynamicDataFactory get_instance();
    /*static*/ DDS::ReturnCode_t delete_instance();

    DynamicData create_data();
    DDS::ReturnCode_t delete_data(in DynamicData data);
};

typedef sequence<long>                  Int32Seq;
typedef sequence<unsigned long>         UInt32Seq;
typedef sequence<short>                 Int16Seq;
typedef sequence<unsigned short>        UInt16Seq;
typedef sequence<long long>             Int64Seq;
typedef sequence<unsigned long long>    UInt64Seq;
typedef sequence<float>                 Float32Seq;
typedef sequence<double>                Float64Seq;
typedef sequence<long double>           Float128Seq;
typedef sequence<char>                  CharSeq;
typedef sequence<wchar>                 WcharSeq;
typedef sequence<boolean>               BooleanSeq;
typedef sequence<octet>                 ByteSeq;

// typedef sequence<string>             StringSeq;
typedef sequence<wstring>               WstringSeq;

/* IDL does not allow the specification of an array without specifying
 * that array's exact bounds. This restriction makes sense when the array
 * is to be sent over a network, but it is overly restrictive when
```

```
 * dealing with local operations. For example, C, C++, Java, and C# all
 * allow array bounds to be unspecified when passing an array to a
 * method (function).
 *
 * Therefore, the following declarations use this convention: any array
 * declaration with dimension UNSPECIFIED_ARRAY_LENGTH, when mapped to a
 * programming language, shall be mapped such that the array bounds are
 * unspecified and may vary from call to call. The
 * UNSPECIFIED_ARRAY_LENGTH constant itself shall be considered an
 * IDL-specific artifact to allow this file to compile; it shall not be
 * reflected in the API.
 */
const unsigned long UNSPECIFIED_ARRAY_LENGTH = 1000;

typedef long                  Int32Array[UNSPECIFIED_ARRAY_LENGTH];
typedef unsigned long         UInt32Array[UNSPECIFIED_ARRAY_LENGTH];
typedef short                 Int16Array[UNSPECIFIED_ARRAY_LENGTH];
typedef unsigned short        UInt16Array[UNSPECIFIED_ARRAY_LENGTH];
typedef long long             Int64Array[UNSPECIFIED_ARRAY_LENGTH];
typedef unsigned long long    UInt64Array[UNSPECIFIED_ARRAY_LENGTH];
typedef float                 Float32Array[UNSPECIFIED_ARRAY_LENGTH];
typedef double                Float64Array[UNSPECIFIED_ARRAY_LENGTH];
typedef long double           Float128Array[UNSPECIFIED_ARRAY_LENGTH];
typedef char                  CharArray[UNSPECIFIED_ARRAY_LENGTH];
typedef wchar                 WcharArray[UNSPECIFIED_ARRAY_LENGTH];
typedef boolean               BooleanArray[UNSPECIFIED_ARRAY_LENGTH];
typedef octet                 ByteArray[UNSPECIFIED_ARRAY_LENGTH];
typedef string                StringArray[UNSPECIFIED_ARRAY_LENGTH];
typedef wstring               WstringArray[UNSPECIFIED_ARRAY_LENGTH];

local interface DynamicData {
    readonly attribute DynamicType type;

    boolean equals(in DynamicData other);
```

```
DDS::ReturnCode_t clear_all_values();
DDS::ReturnCode_t clear_nonkey_values();
DDS::ReturnCode_t clear_value(in ObjectName name);
DDS::ReturnCode_t clear_value_by_id(in MemberId id);
DDS::ReturnCode_t clear_valueat_index(in unsigned long index);

DynamicData loan_value(in ObjectName name);
DynamicData loan_value_by_id(in MemberId id);
DynamicData loan_value_at_index(in unsigned long index);
DDS::ReturnCode_t return_loaned_value(in DynamicData value);

DynamicData clone();

DDS::ReturnCode_t get_int32_value(
    inout long value,
    in ObjectName name);
DDS::ReturnCode_t set_int32_value(
    in ObjectName name,
    in long value);
DDS::ReturnCode_t get_uint32_value(
    inout unsigned long value,
    in ObjectName name);
DDS::ReturnCode_t set_uint32_value(
    in ObjectName name,
    in unsigned long value);
DDS::ReturnCode_t get_int16_value(
    inout short value,
    in ObjectName name);
DDS::ReturnCode_t set_int16_value(
    in ObjectName name,
    in short value);
DDS::ReturnCode_t get_uint16_value(
    inout unsigned short value,
    in ObjectName name);
DDS::ReturnCode_t set_uint16_value(
```

```
        in ObjectName name,
        in unsigned short value);
    DDS::ReturnCode_t get_int64_value(
        inout long long value,
        in ObjectName name);
    DDS::ReturnCode_t set_int64_value(
        in ObjectName name,
        in long long value);
    DDS::ReturnCode_t get_uint64_value(
        inout unsigned long long value,
        in ObjectName name);
    DDS::ReturnCode_t set_uint64_value(
        in ObjectName name,
        in unsigned long long value);
    DDS::ReturnCode_t get_float32_value(
        inout float value,
        in ObjectName name);
    DDS::ReturnCode_t set_float32_value(
        in ObjectName name,
        in float value);
    DDS::ReturnCode_t get_float64_value(
        inout double value,
        in ObjectName name);
    DDS::ReturnCode_t set_float64_value(
        in ObjectName name,
        in double value);
    DDS::ReturnCode_t get_float128_value(
        inout long double value,
        in ObjectName name);
    DDS::ReturnCode_t set_float128_value(
        in ObjectName name,
        in long double value);
    DDS::ReturnCode_t get_char8_value(
        inout char value,
        in ObjectName name);
```

```
DDS::ReturnCode_t set_char8_value(
    in ObjectName name,
    in char value);
DDS::ReturnCode_t get_char32_value(
    inout wchar value,
    in ObjectName name);
DDS::ReturnCode_t set_char32_value(
    in ObjectName name,
    in wchar value);
DDS::ReturnCode_t get_byte_value(
    inout octet value,
    in ObjectName name);
DDS::ReturnCode_t set_byte_value(
    in ObjectName name,
    in octet value);
DDS::ReturnCode_t get_boolean_value(
    inout boolean value,
    in ObjectName name);
DDS::ReturnCode_t set_boolean_value(
    in ObjectName name,
    in boolean value);
DDS::ReturnCode_t get_string_value(
    inout string value,
    in ObjectName name);
DDS::ReturnCode_t set_string_value(
    in ObjectName name,
    in string value);
DDS::ReturnCode_t get_wstring_value(
    inout wstring value,
    in ObjectName name);
DDS::ReturnCode_t set_wstring_value(
    in ObjectName name,
    in wstring value);

DDS::ReturnCode_t get_complex_value(
```

```
        inout DynamicData value,
        in ObjectName name);
    DDS::ReturnCode_t set_complex_value(
        in ObjectName name,
        in DynamicData value);

    DDS::ReturnCode_t get_int32_array_value(
        inout Int32Array value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_int32_array_value(
        in ObjectName name,
        in Int32Array value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_uint32_array_value(
        inout UInt32Array value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_uint32_array_value(
        in ObjectName name,
        in UInt32Array value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_int16_array_value(
        inout Int16Array value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_int16_array_value(
        in ObjectName name,
        in Int16Array value,
        in unsigned long offset,
```

```
    in unsigned long length);
DDS::ReturnCode_t get_uint16_array_value(
    inout UInt16Array value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_uint16_array_value(
    in ObjectName name,
    in UInt16Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_int64_array_value(
    inout Int64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_int64_array_value(
    in ObjectName name,
    in Int64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_uint64_array_value(
    inout UInt64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_uint64_array_value(
    in ObjectName name,
    in UInt64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float32_array_value(
    inout Float32Array value,
    in unsigned long offset,
    inout unsigned long length,
```

```
    in ObjectName name);
DDS::ReturnCode_t set_float32_array_value(
    in ObjectName name,
    in Float32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float64_array_value(
    inout Float64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_float64_array_value(
    in ObjectName name,
    in Float64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float128_array_value(
    inout Float128Array value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_float128_array_value(
    in ObjectName name,
    in Float128Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_char8_array_value(
    inout CharArray value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_char8_array_value(
    in ObjectName name,
    in CharArray value,
    in unsigned long offset,
```

```
        in unsigned long length);
    DDS::ReturnCode_t get_char32_array_value(
        inout WcharArray value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_char32_array_value(
        in ObjectName name,
        in WcharArray value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_byte_array_value(
        inout ByteArray value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_byte_array_value(
        in ObjectName name,
        in ByteArray value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_boolean_array_value(
        inout BooleanArray value,
        in unsigned long offset,
        inout unsigned long length,
        in ObjectName name);
    DDS::ReturnCode_t set_boolean_array_value(
        in ObjectName name,
        in BooleanArray value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_string_array_value(
        inout StringArray value,
        in unsigned long offset,
        inout unsigned long length,
```

```
    in ObjectName name);
DDS::ReturnCode_t set_string_array_value(
    in ObjectName name,
    in StringArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_wstring_array_value(
    inout WstringArray value,
    in unsigned long offset,
    inout unsigned long length,
    in ObjectName name);
DDS::ReturnCode_t set_wstring_array_value(
    in ObjectName name,
    in WstringArray value,
    in unsigned long offset,
    in unsigned long length);

DDS::ReturnCode_t get_int32_seq_value(
    inout Int32Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_int32_seq_value(
    in ObjectName name,
    in Int32Seq value);
DDS::ReturnCode_t get_uint32_seq_value(
    inout UInt32Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_uint32_seq_value(
    in ObjectName name,
    in UInt32Seq value);
DDS::ReturnCode_t get_int16_seq_value(
    inout Int16Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_int16_seq_value(
    in ObjectName name,
    in Int16Seq value);
```

```
DDS::ReturnCode_t get_uint16_seq_value(
    inout UInt16Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_uint16_seq_value(
    in ObjectName name,
    in UInt16Seq value);
DDS::ReturnCode_t get_int64_seq_value(
    inout Int64Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_int64_seq_value(
    in ObjectName name,
    in Int64Seq value);
DDS::ReturnCode_t get_uint64_seq_value(
    inout UInt64Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_uint64_seq_value(
    in ObjectName name,
    in UInt64Seq value);
DDS::ReturnCode_t get_float32_seq_value(
    inout Float32Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_float32_seq_value(
    in ObjectName name,
    in Float32Seq value);
DDS::ReturnCode_t get_float64_seq_value(
    inout Float64Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_float64_seq_value(
    in ObjectName name,
    in Float64Seq value);
DDS::ReturnCode_t get_float128_seq_value(
    inout Float128Seq value,
    in ObjectName name);
DDS::ReturnCode_t set_float128_seq_value(
    in ObjectName name,
```

```
        in Float128Seq value);
    DDS::ReturnCode_t get_char8_seq_value(
        inout CharSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_char8_seq_value(
        in ObjectName name,
        in CharSeq value);
    DDS::ReturnCode_t get_char32_seq_value(
        inout WcharSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_char32_seq_value(
        in ObjectName name,
        in WcharSeq value);
    DDS::ReturnCode_t get_byte_seq_value(
        inout ByteSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_byte_seq_value(
        in ObjectName name,
        in ByteSeq value);
    DDS::ReturnCode_t get_boolean_seq_value(
        inout BooleanSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_boolean_seq_value(
        in ObjectName name,
        in BooleanSeq value);
    DDS::ReturnCode_t get_string_seq_value(
        inout StringSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_string_seq_value(
        in ObjectName name,
        in StringSeq value);
    DDS::ReturnCode_t get_wstring_seq_value(
        inout WstringSeq value,
        in ObjectName name);
    DDS::ReturnCode_t set_wstring_seq_value(
```

```
        in ObjectName name,
        in WstringSeq value);

    DDS::ReturnCode_t get_int32_value_by_id(
        inout long value,
        in MemberId id);
    DDS::ReturnCode_t set_int32_value_by_id(
        in MemberId id,
        in long value);
    DDS::ReturnCode_t get_uint32_value_by_id(
        inout unsigned long value,
        in MemberId id);
    DDS::ReturnCode_t set_uint32_value_by_id(
        in MemberId id,
        in unsigned long value);
    DDS::ReturnCode_t get_int16_value_by_id(
        inout short value,
        in MemberId id);
    DDS::ReturnCode_t set_int16_value_by_id(
        in MemberId id,
        in short value);
    DDS::ReturnCode_t get_uint16_value_by_id(
        inout unsigned short value,
        in MemberId id);
    DDS::ReturnCode_t set_uint16_value_by_id(
        in MemberId id,
        in unsigned short value);
    DDS::ReturnCode_t get_int64_value_by_id(
        inout long long value,
        in MemberId id);
    DDS::ReturnCode_t set_int64_value_by_id(
        in MemberId id,
        in long long value);
    DDS::ReturnCode_t get_uint64_value_by_id(
        inout unsigned long long value,
```

```
    in MemberId id);
DDS::ReturnCode_t set_uint64_value_by_id(
    in MemberId id,
    in unsigned long long value);
DDS::ReturnCode_t get_float32_value_by_id(
    inout float value,
    in MemberId id);
DDS::ReturnCode_t set_float32_value_by_id(
    in MemberId id,
    in float value);
DDS::ReturnCode_t get_float64_value_by_id(
    inout double value,
    in MemberId id);
DDS::ReturnCode_t set_float64_value_by_id(
    in MemberId id,
    in double value);
DDS::ReturnCode_t get_float128_value_by_id(
    inout long double value,
    in MemberId id);
DDS::ReturnCode_t set_float128_value_by_id(
    in MemberId id,
    in long double value);
DDS::ReturnCode_t get_char8_value_by_id(
    inout char value,
    in MemberId id);
DDS::ReturnCode_t set_char8_value_by_id(
    in MemberId id,
    in char value);
DDS::ReturnCode_t get_char32_value_by_id(
    inout Object value,
    in MemberId id);
DDS::ReturnCode_t set_char32_value_by_id(
    in MemberId id,
    in wchar value);
DDS::ReturnCode_t get_byte_value_by_id(
```

```
        inout octet value,
        in MemberId id);
    DDS::ReturnCode_t set_byte_value_by_id(
        in MemberId id,
        in octet value);
    DDS::ReturnCode_t get_boolean_value_by_id(
        inout boolean value,
        in MemberId id);
    DDS::ReturnCode_t set_boolean_value_by_id(
        in MemberId id,
        in boolean value);
    DDS::ReturnCode_t get_string_value_by_id(
        inout string value,
        in MemberId id);
    DDS::ReturnCode_t set_string_value_by_id(
        in MemberId id,
        in string value);
    DDS::ReturnCode_t get_wstring_value_by_id(
        inout wstring value,
        in MemberId id);
    DDS::ReturnCode_t set_wstring_value_by_id(
        in MemberId id,
        in wstring value);

    DDS::ReturnCode_t get_complex_value_by_id(
        inout DynamicData value,
        in MemberId id);
    DDS::ReturnCode_t set_complex_value_by_id(
        in MemberId id,
        in DynamicData value);

    DDS::ReturnCode_t get_int32_array_value_by_id(
        inout Int32Array value,
        in unsigned long offset,
        inout unsigned long length,
```

```
    in MemberId id);
DDS::ReturnCode_t set_int32_array_value_by_id(
    in MemberId id,
    in Int32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_uint32_array_value_by_id(
    inout UInt32Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_uint32_array_value_by_id(
    in MemberId id,
    in UInt32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_int16_array_value_by_id(
    inout Int16Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_int16_array_value_by_id(
    in MemberId id,
    in Int16Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_uint16_array_value_by_id(
    inout UInt16Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_uint16_array_value_by_id(
    in MemberId id,
    in UInt16Array value,
    in unsigned long offset,
```

```
    in unsigned long length);
DDS::ReturnCode_t get_int64_array_value_by_id(
    inout Int64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_int64_array_value_by_id(
    in MemberId id,
    in Int64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_uint64_array_value_by_id(
    inout UInt64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_uint64_array_value_by_id(
    in MemberId id,
    in UInt64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float32_array_value_by_id(
    inout Float32Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_float32_array_value_by_id(
    in MemberId id,
    in Float32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float64_array_value_by_id(
    inout Float64Array value,
    in unsigned long offset,
    inout unsigned long length,
```

```
    in MemberId id);
DDS::ReturnCode_t set_float64_array_value_by_id(
    in MemberId id,
    in Float64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float128_array_value_by_id(
    inout Float128Array value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_float128_array_value_by_id(
    in MemberId id,
    in Float128Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_char8_array_value_by_id(
    inout CharArray value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_char8_array_value_by_id(
    in MemberId id,
    in CharArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_char32_array_value_by_id(
    inout WcharArray value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_char32_array_value_by_id(
    in MemberId id,
    in WcharArray value,
    in unsigned long offset,
```

```
    in unsigned long length);
DDS::ReturnCode_t get_byte_array_value_by_id(
    inout ByteArray value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_byte_array_value_by_id(
    in MemberId id,
    in ByteArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_boolean_array_value_by_id(
    inout BooleanArray value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_boolean_array_value_by_id(
    in MemberId id,
    in BooleanArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_string_array_value_by_id(
    inout StringArray value,
    in unsigned long offset,
    inout unsigned long length,
    in MemberId id);
DDS::ReturnCode_t set_string_array_value_by_id(
    in MemberId id,
    in StringArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_wstring_array_value_by_id(
    inout WstringArray value,
    in unsigned long offset,
    inout unsigned long length,
```

```
        in MemberId id);
    DDS::ReturnCode_t set_wstring_array_value_by_id(
        in MemberId id,
        in WstringArray value,
        in unsigned long offset,
        in unsigned long length);

    DDS::ReturnCode_t get_int32_seq_value_by_id(
        inout Int32Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_int32_seq_value_by_id(
        in MemberId id,
        in Int32Seq value);
    DDS::ReturnCode_t get_uint32_seq_value_by_id(
        inout UInt32Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_uint32_seq_value_by_id(
        in MemberId id,
        in UInt32Seq value);
    DDS::ReturnCode_t get_int16_seq_value_by_id(
        inout Int16Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_int16_seq_value_by_id(
        in MemberId id,
        in Int16Seq value);
    DDS::ReturnCode_t get_uint16_seq_value_by_id(
        inout UInt16Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_uint16_seq_value_by_id(
        in MemberId id,
        in UInt16Seq value);
    DDS::ReturnCode_t get_int64_seq_value_by_id(
        inout Int64Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_int64_seq_value_by_id(
```

```
        in MemberId id,
        in Int64Seq value);
    DDS::ReturnCode_t get_uint64_seq_value_by_id(
        inout UInt64Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_uint64_seq_value_by_id(
        in MemberId id,
        in UInt64Seq value);
    DDS::ReturnCode_t get_float32_seq_value_by_id(
        inout Float32Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_float32_seq_value_by_id(
        in MemberId id,
        in Float32Seq value);
    DDS::ReturnCode_t get_float64_seq_value_by_id(
        inout Float64Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_float64_seq_value_by_id(
        in MemberId id,
        in Float64Seq value);
    DDS::ReturnCode_t get_float128_seq_value_by_id(
        inout Float128Seq value,
        in MemberId id);
    DDS::ReturnCode_t set_float128_seq_value_by_id(
        in MemberId id,
        in Float128Seq value);
    DDS::ReturnCode_t get_char8_seq_value_by_id(
        inout CharSeq value,
        in MemberId id);
    DDS::ReturnCode_t set_char8_seq_value_by_id(
        in MemberId id,
        in CharSeq value);
    DDS::ReturnCode_t get_char32_seq_value_by_id(
        inout WcharSeq value,
        in MemberId id);
```

```
DDS::ReturnCode_t set_char32_seq_value_by_id(
    in MemberId id,
    in WcharSeq value);
DDS::ReturnCode_t get_byte_seq_value_by_id(
    inout ByteSeq value,
    in MemberId id);
DDS::ReturnCode_t set_byte_seq_value_by_id(
    in MemberId id,
    in ByteSeq value);
DDS::ReturnCode_t get_boolean_seq_value_by_id(
    inout BooleanSeq value,
    in MemberId id);
DDS::ReturnCode_t set_boolean_seq_value_by_id(
    in MemberId id,
    in BooleanSeq value);
DDS::ReturnCode_t get_string_seq_value_by_id(
    inout StringSeq value,
    in MemberId id);
DDS::ReturnCode_t set_string_seq_value_by_id(
    in MemberId id,
    in StringSeq value);
DDS::ReturnCode_t get_wstring_seq_value_by_id(
    inout WstringSeq value,
    in MemberId id);
DDS::ReturnCode_t set_wstring_seq_value_by_id(
    in MemberId id,
    in WstringSeq value);

DDS::ReturnCode_t get_int32_value_at_index(
    inout long value,
    in unsigned long index);
DDS::ReturnCode_t set_int32_value_at_index(
    in unsigned long index,
    in long value);
DDS::ReturnCode_t get_uint32_value_at_index(
```

```
        inout unsigned long value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint32_value_at_index(
        in unsigned long index,
        in unsigned long value);
    DDS::ReturnCode_t get_int16_value_at_index(
        inout short value,
        in unsigned long index);
    DDS::ReturnCode_t set_int16_value_at_index(
        in unsigned long index,
        in short value);
    DDS::ReturnCode_t get_uint16_value_at_index(
        inout unsigned short value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint16_value_at_index(
        in unsigned long index,
        in unsigned short value);
    DDS::ReturnCode_t get_int64_value_at_index(
        inout long long value,
        in unsigned long index);
    DDS::ReturnCode_t set_int64_value_at_index(
        in unsigned long index,
        in long long value);
    DDS::ReturnCode_t get_uint64_value_at_index(
        inout unsigned long long value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint64_value_at_index(
        in unsigned long index,
        in unsigned long long value);
    DDS::ReturnCode_t get_float32_value_at_index(
        inout float value,
        in unsigned long index);
    DDS::ReturnCode_t set_float32_value_at_index(
        in unsigned long index,
        in float value);
```

```
DDS::ReturnCode_t get_float64_value_at_index(
    inout double value,
    in unsigned long index);
DDS::ReturnCode_t set_float64_value_at_index(
    in unsigned long index,
    in double value);
DDS::ReturnCode_t get_float128_value_at_index(
    inout long double value,
    in unsigned long index);
DDS::ReturnCode_t set_float128_value_at_index(
    in unsigned long index,
    in long double value);
DDS::ReturnCode_t get_char8_value_at_index(
    inout char value,
    in unsigned long index);
DDS::ReturnCode_t set_char8_value_at_index(
    in unsigned long index,
    in char value);
DDS::ReturnCode_t get_char32_value_at_index(
    inout wchar value,
    in unsigned long index);
DDS::ReturnCode_t set_char32_value_at_index(
    in unsigned long index,
    in wchar value);
DDS::ReturnCode_t get_byte_value_at_index(
    inout octet value,
    in unsigned long index);
DDS::ReturnCode_t set_byte_value_at_index(
    in unsigned long index,
    in octet value);
DDS::ReturnCode_t get_boolean_value_at_index(
    inout boolean value,
    in unsigned long index);
DDS::ReturnCode_t set_boolean_value_at_index(
    in unsigned long index,
```

```
    in boolean value);
DDS::ReturnCode_t get_string_value_at_index(
    inout string value,
    in unsigned long index);
DDS::ReturnCode_t set_string_value_at_index(
    in unsigned long index,
    in string value);
DDS::ReturnCode_t get_wstring_value_at_index(
    inout wstring value,
    in unsigned long index);
DDS::ReturnCode_t set_wstring_value_at_index(
    in unsigned long index,
    in wstring value);

DDS::ReturnCode_t get_int32_array_value_at_index(
    inout Int32Array value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_int32_array_value_at_index(
    in unsigned long index,
    in Int32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_uint32_array_value_at_index(
    inout UInt32Array value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_uint32_array_value_at_index(
    in unsigned long index,
    in UInt32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_int16_array_value_at_index(
```

```
        inout Int16Array value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_int16_array_value_at_index(
        in unsigned long index,
        in Int16Array value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_uint16_array_value_at_index(
        inout UInt16Array value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_uint16_array_value_at_index(
        in unsigned long index,
        in UInt16Array value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_int64_array_value_at_index(
        inout Int64Array value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_int64_array_value_at_index(
        in unsigned long index,
        in Int64Array value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_uint64_array_value_at_index(
        inout UInt64Array value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_uint64_array_value_at_index(
```

```
    in unsigned long index,
    in UInt64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float32_array_value_at_index(
    inout Float32Array value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_float32_array_value_at_index(
    in unsigned long index,
    in Float32Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float64_array_value_at_index(
    inout Float64Array value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_float64_array_value_at_index(
    in unsigned long index,
    in Float64Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_float128_array_value_at_index(
    inout Float128Array value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_float128_array_value_at_index(
    in unsigned long index,
    in Float128Array value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_char8_array_value_at_index(
```

```
    inout CharArray value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_char8_array_value_at_index(
    in unsigned long index,
    in CharArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_char32_array_value_at_index(
    inout WcharArray value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_char32_array_value_at_index(
    in unsigned long index,
    in WcharArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_byte_array_value_at_index(
    inout ByteArray value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_byte_array_value_at_index(
    in unsigned long index,
    in ByteArray value,
    in unsigned long offset,
    in unsigned long length);
DDS::ReturnCode_t get_boolean_array_value_at_index(
    inout BooleanArray value,
    in unsigned long offset,
    inout unsigned long length,
    in unsigned long index);
DDS::ReturnCode_t set_boolean_array_value_at_index(
```

```
        in unsigned long index,
        in BooleanArray value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_string_array_value_at_index(
        inout StringArray value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_string_array_value_at_index(
        in unsigned long index,
        in StringArray value,
        in unsigned long offset,
        in unsigned long length);
    DDS::ReturnCode_t get_wstring_array_value_at_index(
        inout WstringArray value,
        in unsigned long offset,
        inout unsigned long length,
        in unsigned long index);
    DDS::ReturnCode_t set_wstring_array_value_at_index(
        in unsigned long index,
        in WstringArray value,
        in unsigned long offset,
        in unsigned long length);

    DDS::ReturnCode_t get_int32_seq_value_at_index(
        inout Int32Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_int32_seq_value_at_index(
        in unsigned long index,
        in Int32Seq value);
    DDS::ReturnCode_t get_uint32_seq_value_at_index(
        inout UInt32Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint32_seq_value_at_index(
```

```
        in unsigned long index,
        in UInt32Seq value);
    DDS::ReturnCode_t get_int16_seq_value_at_index(
        inout Int16Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_int16_seq_value_at_index(
        in unsigned long index,
        in Int16Seq value);
    DDS::ReturnCode_t get_uint16_seq_value_at_index(
        inout UInt16Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint16_seq_value_at_index(
        in unsigned long index,
        in UInt16Seq value);
    DDS::ReturnCode_t get_int64_seq_value_at_index(
        inout Int64Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_int64_seq_value_at_index(
        in unsigned long index,
        in Int64Seq value);
    DDS::ReturnCode_t get_uint64_seq_value_at_index(
        inout UInt64Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_uint64_seq_value_at_index(
        in unsigned long index,
        in UInt64Seq value);
    DDS::ReturnCode_t get_float32_seq_value_at_index(
        inout Float32Seq value,
        in unsigned long index);
    DDS::ReturnCode_t set_float32_seq_value_at_index(
        in unsigned long index,
        in Float32Seq value);
    DDS::ReturnCode_t get_float64_seq_value_at_index(
        inout Float64Seq value,
        in unsigned long index);
```

```
DDS::ReturnCode_t set_float64_seq_value_at_index(
    in unsigned long index,
    in Float64Seq value);
DDS::ReturnCode_t get_float128_seq_value_at_index(
    inout Float128Seq value,
    in unsigned long index);
DDS::ReturnCode_t set_float128_seq_value_at_index(
    in unsigned long index,
    in Float128Seq value);
DDS::ReturnCode_t get_char8_seq_value_at_index(
    inout CharSeq value,
    in unsigned long index);
DDS::ReturnCode_t set_char8_seq_value_at_index(
    in unsigned long index,
    in CharSeq value);
DDS::ReturnCode_t get_char32_seq_value_at_index(
    inout WcharSeq value,
    in unsigned long index);
DDS::ReturnCode_t set_char32_seq_value_at_index(
    in unsigned long index,
    in WcharSeq value);
DDS::ReturnCode_t get_byte_seq_value_at_index(
    inout ByteSeq value,
    in unsigned long index);
DDS::ReturnCode_t set_byte_seq_value_at_index(
    in unsigned long index,
    in ByteSeq value);
DDS::ReturnCode_t get_boolean_seq_value_at_index(
    inout BooleanSeq value,
    in unsigned long index);
DDS::ReturnCode_t set_boolean_seq_value_at_index(
    in unsigned long index,
    in BooleanSeq value);
DDS::ReturnCode_t get_string_seq_value_at_index(
    inout StringSeq value,
```

```
            in unsigned long index);
        DDS::ReturnCode_t set_string_seq_value_at_index(
            in unsigned long index,
            in StringSeq value);
        DDS::ReturnCode_t get_wstring_seq_value_at_index(
            inout WstringSeq value,
            in unsigned long index);
        DDS::ReturnCode_t set_wstring_seq_value_at_index(
            in unsigned long index,
            in WstringSeq value);
    };  // local interface DynamicData
};  // end module DDS
```

Annex D: DDS Built-in Topic Data Types

Previously, the standard DDS type system (based solely on IDL prior to the extensions introduced by this specification) was insufficiently rich to represent the built-in topic data to the level specified by DDS [DDS] and RTPS [RTPS]. This specification remedies this situation. The following are expanded definitions of the built-in topic data types that contain all of the meta-data necessary to represent them as defined by the existing DDS and RTPS specifications.

```
module DDS {
    @Nested
    struct BuiltinTopicKey_t {
        long value[4];
    };

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

    @Nested
    struct DeadlineQosPolicy {
        Duration_t period;
    };

    enum DestinationOrderQosPolicyKind {
        BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
        BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
    };

    @Nested
    struct DestinationOrderQosPolicy {
        DestinationOrderQosPolicyKind kind;
    };

    enum DurabilityQosPolicyKind {
```

```
    VOLATILE_DURABILITY_QOS,
    TRANSIENT_LOCAL_DURABILITY_QOS,
    TRANSIENT_DURABILITY_QOS,
    PERSISTENT_DURABILITY_QOS
};

@Nested
struct DurabilityQosPolicy {
    DurabilityQosPolicyKind kind;
};

enum HistoryQosPolicyKind {
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};

@Nested
struct HistoryQosPolicy {
    HistoryQosPolicyKind kind;
    long depth;
};

@Nested
struct DurabilityServiceQosPolicy {
    Duration_t service_cleanup_delay;
    HistoryQosPolicyKind history_kind;
    long history_depth;
    long max_samples;
    long max_instances;
    long max_samples_per_instance;
};

@Nested
struct GroupDataQosPolicy {
    sequence<octet> value;
```

```
};

@Nested
struct LatencyBudgetQosPolicy {
    Duration_t duration;
};

@Nested
struct LifespanQosPolicy {
    Duration_t duration;
};

enum LivelinessQosPolicyKind {
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS
};

@Nested
struct LivelinessQosPolicy {
    LivelinessQosPolicyKind kind;
    Duration_t lease_duration;
};

enum OwnershipQosPolicyKind {
    SHARED_OWNERSHIP_QOS,
    EXCLUSIVE_OWNERSHIP_QOS
};

@Nested
struct OwnershipQosPolicy {
    OwnershipQosPolicyKind kind;
};

@Nested
```

```
struct OwnershipStrengthQosPolicy {
    long value;
};

@Nested
struct PartitionQosPolicy {
    sequence<string> name;
};

enum PresentationQosPolicyAccessScopeKind {
    INSTANCE_PRESENTATION_QOS,
    TOPIC_PRESENTATION_QOS,
    GROUP_PRESENTATION_QOS
};

@Nested
struct PresentationQosPolicy {
    PresentationQosPolicyAccessScopeKind access_scope;
    boolean coherent_access;
    boolean ordered_access;
};

enum ReliabilityQosPolicyKind {
    BEST_EFFORT_RELIABILITY_QOS,
    RELIABLE_RELIABILITY_QOS
};

@Nested
struct ReliabilityQosPolicy {
    ReliabilityQosPolicyKind kind;
    Duration_t max_blocking_time;
};
```

```
@Nested
struct ResourceLimitsQosPolicy {
    long max_samples;
    long max_instances;
    long max_samples_per_instance;
};

@Nested
struct TimeBasedFilterQosPolicy {
    Duration_t minimum_separation;
};

@Nested
struct TopicDataQosPolicy {
    sequence<octet> value;
};

@Nested
struct TransportPriorityQosPolicy {
    long value;
};

@Nested
struct UserDataQosPolicy {
    sequence<octet> value;
};

@Extensibility(MUTABLE_EXTENSIBILITY)
struct ParticipantBuiltinTopicData {
    @ID(0x0050) @Key BuiltinTopicKey_t key;
    @ID(0x002C)      UserDataQosPolicy user_data;
};

typedef short DataRepresentationId_t;
```

```
const DataRepresentationId_t XCDR_REPRESENTATION = 0;
const DataRepresentationId_t XML_REPRESENTATION  = 1;

typedef sequence<DataRepresentationId_t> DataRepresentationIdSeq;

const QosPolicyId_t DATA_REPRESENTATION_QOS_POLICY_ID = 23;
const string DATA_REPRESENTATION_QOS_POLICY_NAME = "DataRepresentation";

@Nested
struct DataRepresentationQosPolicy {
    DataRepresentationIdSeq value;
};

@BitBound(16)
enum TypeConsistencyKind {
    EXACT_TYPE_TYPE_CONSISTENCY,
    EXACT_NAME_TYPE_CONSISTENCY,
    DECLARED_TYPE_CONSISTENCY,
    ASSIGNABLE_TYPE_CONSISTENCY
};

const QosPolicyId_t TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_ID = 24;
const string TYPE_CONSISTENCY_ENFORCEMENT_QOS_POLICY_NAME =
    "TypeConsistencyEnforcement";

@Nested
struct TypeConsistencyEnforcementQosPolicy {
    TypeConsistencyKind kind;
};

@Extensibility(MUTABLE_EXTENSIBILITY)
struct TopicBuiltinTopicData {
    @ID(0x005A) @Key BuiltinTopicKey_t key;
    @ID(0x0005)      string<256> name;
    @ID(0x0007)      string<256> type_name;
```

```
    @ID(0x0072) @Optional   DDS::TypeObject type;
    @ID(0x001D)      DurabilityQosPolicy durability;
    @ID(0x001E)      DurabilityServiceQosPolicy durability_service;
    @ID(0x0023)      DeadlineQosPolicy deadline;
    @ID(0x0027)      LatencyBudgetQosPolicy latency_budget;
    @ID(0x001B)      LivelinessQosPolicy liveliness;
    @ID(0x001A)      ReliabilityQosPolicy reliability;
    @ID(0x0049)      TransportPriorityQosPolicy transport_priority;
    @ID(0x002B)      LifespanQosPolicy lifespan;
    @ID(0x0025)      DestinationOrderQosPolicy destination_order;
    @ID(0x0040)      HistoryQosPolicy history;
    @ID(0x0041)      ResourceLimitsQosPolicy resource_limits;
    @ID(0x001F)      OwnershipQosPolicy ownership;
    @ID(0x002E)      TopicDataQosPolicy topic_data;
    @ID(0x0073)      DataRepresentationQosPolicy representation;
    @ID(0x0074)      TypeConsistencyEnforcementQosPolicy type_consistency;
};

@Extensibility(MUTABLE_EXTENSIBILITY)
struct PublicationBuiltinTopicData {
    @ID(0x005A) @Key BuiltinTopicKey_t key;
    @ID(0x0050)      BuiltinTopicKey_t participant_key;
    @ID(0x0005)      string<256> topic_name;
    @ID(0x0007)      string<256> type_name;
    @ID(0x0072) @Optional   DDS::TypeObject type;
    @ID(0x001D)      DurabilityQosPolicy durability;
    @ID(0x001E)      DurabilityServiceQosPolicy durability_service;
    @ID(0x0023)      DeadlineQosPolicy deadline;
    @ID(0x0027)      LatencyBudgetQosPolicy latency_budget;
    @ID(0x001B)      LivelinessQosPolicy liveliness;
    @ID(0x001A)      ReliabilityQosPolicy reliability;
    @ID(0x002B)      LifespanQosPolicy lifespan;
    @ID(0x002C)      UserDataQosPolicy user_data;
    @ID(0x001F)      OwnershipQosPolicy ownership;
    @ID(0x0006)      OwnershipStrengthQosPolicy ownership_strength;
```

```
        @ID(0x0025)      DestinationOrderQosPolicy destination_order;
        @ID(0x0021)      PresentationQosPolicy presentation;
        @ID(0x0029)      PartitionQosPolicy partition;
        @ID(0x002E)      TopicDataQosPolicy topic_data;
        @ID(0x002D)      GroupDataQosPolicy group_data;
        @ID(0x0073)      DataRepresentationQosPolicy representation;
        @ID(0x0074)      TypeConsistencyEnforcementQosPolicy type_consistency;
    };

    @Extensibility(MUTABLE_EXTENSIBILITY)
    struct SubscriptionBuiltinTopicData {
        @ID(0x005A) @Key BuiltinTopicKey_t key;
        @ID(0x0050)      BuiltinTopicKey_t participant_key;
        @ID(0x0005)      string<256> topic_name;
        @ID(0x0007)      string<256> type_name;
        @ID(0x0072) @Optional   DDS::TypeObject type;
        @ID(0x001D)      DurabilityQosPolicy durability;
        @ID(0x0023)      DeadlineQosPolicy deadline;
        @ID(0x0027)      LatencyBudgetQosPolicy latency_budget;
        @ID(0x001B)      LivelinessQosPolicy liveliness;
        @ID(0x001A)      ReliabilityQosPolicy reliability;
        @ID(0x001F)      OwnershipQosPolicy ownership;
        @ID(0x0025)      DestinationOrderQosPolicy destination_order;
        @ID(0x002C)      UserDataQosPolicy user_data;
        @ID(0x0004)      TimeBasedFilterQosPolicy time_based_filter;
        @ID(0x0021)      PresentationQosPolicy presentation;
        @ID(0x0029)      PartitionQosPolicy partition;
        @ID(0x002E)      TopicDataQosPolicy topic_data;
        @ID(0x002D)      GroupDataQosPolicy group_data;
        @ID(0x0073)      DataRepresentationQosPolicy representation;
        @ID(0x0074)      TypeConsistencyEnforcementQosPolicy type_consistency;
    };
};  // end module DDS
```

Annex E: Built-in Types

DDS shall provide a few very types preregistered “out of the box” to allow users to address certain simple use cases without the need for code generation, dynamic type definition, or type registration. These types are defined below[12].

```
module DDS {
    struct _String {
        string value;
    };

    interface StringDataWriter : DataWriter {
        /* This interface shall instantiate the type FooDataWriter defined by
         * the DDS specification where "Foo" is an unbounded string.
         */
    };

    interface StringDataReader : DataReader {
        /* This interface shall instantiate the type FooDataReader defined by
         * the DDS specification where "Foo" is an unbounded string.
         */
    };


    struct KeyedString {
        @Key string key;
        string value;
    };
    typedef sequence<KeyedString> KeyedStringSeq;

    interface KeyedStringDataWriter : DataWriter {
        /* This interface shall instantiate the type FooDataWriter defined by
         * the DDS specification where "Foo" is KeyedString. It also defines
```

[12] The leading underscore in the declaration of the `String` structure is necessary to prevent collision with the IDL keyword “string.” According to the IDL specification, it is treated as an escaping character and is not considered part of the identifier.

```
 * the operations below.
 */
InstanceHandle_t register_instance_w_key(
    in string key);
InstanceHandle_t register_instance_w_key_w_timestamp(
    in string key,
    in Time_t source_timestamp);

ReturnCode_t unregister_instance_w_key(
    in string key,
    in InstanceHandle_t handle);
ReturnCode_t unregister_instance_w_key_w_timestamp(
    in string key,
    in InstanceHandle_t handle,
    in Time_t source_timestamp);

ReturnCode_t write_string_w_key(
    in string key,
    in string str,
    in InstanceHandle_t handle);
ReturnCode_t write_string_w_key_w_timestamp(
    in string key,
    in string str,
    in InstanceHandle_t handle,
    in Time_t source_timestamp);

ReturnCode_t dispose_w_key(
    in string key,
    in InstanceHandle_t instance_handle);
ReturnCode_t dispose_w_key_w_timestamp(
    in string key,
    in InstanceHandle_t instance_handle,
    in Time_t source_timestamp);

ReturnCode_t get_key_value_w_key(
```

```
        inout string key,
        in InstanceHandle_t handle);

    InstanceHandle_t lookup_instance_w_key(
        in string key);
};

interface KeyedStringDataReader : DataReader {
    /* This interface shall instantiate the type FooDataReader defined by
     * the DDS specification where "Foo" is KeyedString.
     */
};


struct Bytes {
    sequence<octet> value;
};
typedef sequence<Bytes> BytesSeq;

interface BytesDataWriter : DataWriter {
    /* This interface shall instantiate the type FooDataWriter defined by
     * the DDS specification where "Foo" is an unbounded sequence of
     * bytes (octets). It also defines the operations below.
     */
    ReturnCode_t write_string_w_bytes(
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle);
    ReturnCode_t write_string_w_bytes_w_timestamp(
          in ByteArray bytes,
          in long offset,
          in long length,
          in InstanceHandle_t handle,
          in Time_t source_timestamp);
```

```
};

interface BytesDataReader : DataReader {
    /* This interface shall instantiate the type FooDataReader defined by
     * the DDS specification where "Foo" is Bytes.
     */
};


struct KeyedBytes {
    @Key string key;
    sequence<octet> value;
};
typedef sequence<KeyedBytes> KeyedBytesSeq;

interface KeyedBytesDataWriter : DataWriter {
    /* This interface shall instantiate the type FooDataWriter defined by
     * the DDS specification where "Foo" is KeyedBytes. It also defines
     * It also defines the operations below.
     */
    InstanceHandle_t register_instance_w_key(
        in string key);
    InstanceHandle_t register_instance_w_key_w_timestamp(
        in string key,
        in Time_t source_timestamp);

    ReturnCode_t unregister_instance_w_key(
        in string key,
        in InstanceHandle_t handle);
    ReturnCode_t unregister_instance_w_key_w_timestamp(
        in string key,
        in InstanceHandle_t handle,
        in Time_t source_timestamp);

    ReturnCode_t write_bytes_w_key(
```

```
            in ByteArray bytes,
            in long offset,
            in long length,
            in InstanceHandle_t handle);
        ReturnCode_t write_bytes_w_key_w_timestamp(
            in ByteArray bytes,
            in long offset,
            in long length,
            in InstanceHandle_t handle,
            in Time_t source_timestamp);

        ReturnCode_t dispose_w_key(
            in string key,
            in InstanceHandle_t instance_handle);
        ReturnCode_t dispose_w_key_w_timestamp(
            in string key,
            in InstanceHandle_t instance_handle,
            in Time_t source_timestamp);

        ReturnCode_t get_key_value_w_key(
            inout string key,
            in InstanceHandle_t handle);

        InstanceHandle_t lookup_instance_w_key(
            in string key);
    };

    interface KeyedBytesDataReader : DataReader {
        /* This interface shall instantiate the type FooDataReader defined by
         * the DDS specification where "Foo" is KeyedBytes.
         */
    };
};  // end module DDS
```