Date: November 2010

Java 5 Language PSM for DDS

Revised Submission 2

OMG Document Number:	**mars/2010-11-03**
Associated File(s):	**omgdds.jar (mars/2010-11-04)**
	omgdds_src.zip (mars/2010-11-05)

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.

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.)

Table of Contents

Preface

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 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
- Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- 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 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
Building A, 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

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

PART I - Introduction

1 Response Details

1.1 OMG RFP Response

This specification is submitted in response to the Java 5 Language PSM for DDS RFP issued in December 2009 with OMG document number mars/09-12-16.

1.2 Submitters

The following OMG members are making this submission:

- Real-Time Innovations, Inc. (RTI)
- PrismTech

1.3 Copyright Waiver

1.4 Contacts

- Real-Time Innovations – Rick Warren rick.warren@rti.com
- PrismTech – Angelo Corsaro angelo.corsaro@prismtech.com

1.5 Problem Statement

The DDS specification [DDS] uses an MDA approach to describe the API and behavior of the middleware. The specification defines a Platform Independent Model (PIM) and a Platform Specific Model (PSM).

The PIM uses UML to specify the interfaces the application can use to participate in a DDS system, including the creation of DDS Entities (`DomainParticipants`, `Topics`, `Publishers`, `Subscribers`, `DataWriters`, and `DataReaders`), the configuration of their QoS, the sending and reception of data, the reception of feedback and notifications from the middleware, etc. The PIM is independent of the concrete programming language (e.g. C++, Java)

used by the application.

A typical application programmer, however, does not use UML directly. Rather, he or she must use the API bindings in their programming language of choice (e.g. C++, Java, C, etc.). In the DDS specification, these language mappings are defined using Platform-Specific Models (PSMs).

For convenience, instead of defining a PSM for each target programming language (C++, Java, C, C#, etc.), the DDS specification version 1.2 defines a single PSM to the IDL language. While IDL is not a "programming language" *per se*, the OMG has defined mappings from IDL to most programming languages, and therefore this process indirectly defines PSMs in most programming languages.

However, the use of IDL as an intermediate language in combination with the IDL mappings to concrete programming languages presents several problems. Specifically, in the case of Java 5:

- IDL does not support many natural Java 5 programming-language features, such as generic types.
- The IDL mapping does not use the standard Java 5 containers (*e.g.* `java.util.List`).
- In general, the Java community has a very strong culture of naming conventions and coding conventions. The IDL PSM and the Java API generated from it violate these conventions, which make DDS feel foreign and unfamiliar to Java programmers.
- APIs derived from the IDL mapping do not support Java Beans, `java.io.Serializable`, and other built-in capabilities of the Java platform.
- The IDL mapping to Java prevents crucial performance-enabling features of the DDS PIM, including loaning samples from a `DataReader`[1].

The end result is a Java API that is artificially and unnecessarily limited in terms of both its performance and its usability.

With the increasing adoption of DDS for the integration of large distributed systems, it is desirable to define native language interfaces that would facilitate the use of the standard and increase vendor portability.

Extending IDL would not accomplish the same objectives. Currently, IDL offers a minimal set of features that can reasonably supported on most languages and provide a uniform interface across languages. On the other hand, the purpose of the native programming language PSMs is to be able to exploit the most natural features of each language. If IDL were to be extended to support

[1] The `DataReader` `read()` and `take()` operations, as specified in the DDS PIM, take in two sequences, which they fill with sample data and meta-data. The output lengths of these sequences correspond to the number of samples retrieved. The IDL-to-Java mapping for sequences maps them to arrays, which has two problems: (*1*) There is a conflation between the current and maximum lengths of the sequence, which requires the operations to allocate new arrays (and the garbage collector to reclaim the old ones), and reassign all the pointers in them, any time the logical length changes. (*2*) There is no abstract interface a DDS vendor can implement in terms of its efficient internal data structures. Instead, it must copy samples into the arrays one at a time. Furthermore, it must find some other place to store the state of the loan. The result of both of these limitations is an implementation that uses more memory and is substantially lower performing.

all of these features, we would end up with an IDL that combines the features of all languages.

1.6 Overview of this Specification

This specification defines a platform-specific model (PSM) for the OMG Data Distribution Service for Real-Time Systems (DDS). It is organized according to the modules defined by the DDS PIM and the types and operations defined within them. As background for this specification, readers may wish to review the DDS specification, which complements this specification.

Of primary concern to applications deploying DDS technology is the ability to integrate applications running on heterogeneous platforms, which may differ both in the programming language use as well as the underlying operating system. This goal is not compromised by the introduction of additional programming language PSMs.

More specifically:

- The new PSM is functionally equivalent to the IDL PSM. While the syntax may change, the application is able to accomplish exactly the same tasks as using the existing PSM.
- Applications programmed using different PSMs can interoperate transparently, *i.e.*, an application that communicates with another one need have no awareness, other than informational, of what PSM the peer application is using.
- The new PSM is derivable from the PIM by following a fairly small set of rules, which are described below.
- The new PSM uses common idioms for the Java language, *e.g.*, generic interfaces.

The new PSM is designed such that it can co-exist with other technologies; for example, it does not interfere with the namespaces used by other OMG specifications or commonly used packages.

1.7 Design Rationale

This PSM has been designed according to the following principles:

- **Ease of use**. This PSM provides simplified method overloads to make DDS-standard operations easier to call. In addition, it aligns with common Java technologies and naming conventions in order to make it easier to use DDS alongside Java Beans, Collections, and Serialization.
- **Run-time portability**. Like the JMS API ([JMS]), this DDS PSM allows applications to be compiled against standard interfaces and then deployed against an arbitrary implementation of those interfaces. It is even possible for a single application to use multiple DDS implementations within a single JVM instance, making it possible—for example—to develop application-level bridging capabilities or to integrate components that were developed targeting different DDS implementations.
- **Dynamic loading and unloading**. This PSM does not require the implementer to maintain any static state. Implementations that do not use static state can be dynamically

loaded and unloaded much more easily; this property is helpful when the target deployment environment is a managed container of some kind, such as an OSGi runtime or Java EE application server. Such a container may or may not use Java class loaders in a way that allows static state to be unloaded. Furthermore, such a container may be used to compose applications of independently-developed components, some of which may have been developed with different DDS implementations (or different versions of the "same" DDS implementation).

- **Performance**. Existing Java communication APIs, such as the DDS IDL PSM or JMS, artificially limit performance by forcing implementations to unnecessarily allocate new objects on the critical send and receive paths. This PSM avoids this pitfall, allowing for much higher-performing implementations.

1.8 Statement of Proof of Concept

The submitter currently offers a number of DDS language PSMs as a commercial off-the-shelf offering. These all interoperate seamlessly—with one another and with DDS PSM implementations from other DDS vendors. This degree of decoupling between the PSM of the programming language API and the network protocol provides a high degree of confidence that this additional PSM will not impact interoperability.

In addition, the submitter has provided the Java interfaces that comprise this submission as open source for public comment. These are available in the DataDistrib4J project hosted on Google Code: http://code.google.com/p/datadistrib4j/.

1.9 Resolution of RFP Requirements and Requests

The specification resolves the following mandatory requirements:

Table 1. Mandatory RFP Requirements

Requirement	Description	Reference	Remarks
6.5.1	Proposals shall provide a PSM, derived from the DDS PIM, targeting the Java 5 language. It shall also include the rules by which this PSM was derived from that PIM. (These rules may be expressed either in prose or in a formal language.)	**Section 8**	**Satisfied**. Transformations are expressed in prose.
6.5.1	The proposed PSM shall implement the DDS PIM in its entirety.	**Section 8**	**Partially satisfied**; the PSM implements only the DCPS portion of the DDS PIM. It omits DLRL.
6.5.3	The use of the proposed PSM shall have no impact on interoperability (for a single vendor or between vendor products). An application that	—	**Satisfied**: the PSM does not refer to or impact the network protocol used by an implementation in any way.

	communicates with another one shall not require any awareness of the PSM used by the peer application. (The proposal *need not* support the use of multiple PSMs within the same Java virtual machine instance.)		Note: the PSM does not preclude the use of multiple PSMs within the same JVM instance.
6.5.4	Proposals shall not introduce new container types without strong justification and shall instead use instead Java-standard types, such as `java.util.List` or `java.lang.String`.	**Section 8**	**Satisfied**: the PSM uses the Java SE-standard `Collection`, `List`, and `Set` interfaces.
6.5.5	Proposals shall not modify the existing language used for user-defined data types associated with published and subscribed Topics (data-type definitions) or the language mapping of those data types.	**—**	**Satisfied**. This PSM does not invalidate or modify any of the Type Representations or Language Bindings specified by [DDS-XTypes]. However, it does introduce an additional Type Representation: the Java Type Representation.
6.5.6	Proposals shall allow applications to choose among implementations of the PSM at runtime. That is, an application compiled against the proposed APIs shall be able to use at runtime any DDS implementation conformant to those APIs without modification.	**Section 8.2.1**	**Satisfied**; applications may choose one or more run-time implementations by means of a Java system property.
6.5.7	Proposals shall not introduce dependencies on any other communications middleware technology—including, but not limited to, CORBA or JMS—as doing so would necessarily modify the meaning of conformance for DDS and/or the other technology.	**—**	**Satisfied**.
6.5.8	Proposals shall indicate which Java Edition(s) (*i.e.*, Java Micro Edition, Java Standard Edition, and Java Enterprise Edition) they target.	**Section 7.2**	**Satisfied**: the specification targets Java SE.
6.5.9	Proposals shall be designed such that they can co-exist with other technologies. For example, they	**Section 8.1.1**	**Satisfied**: all types defined by this specification exist in packages named with the

	should not interfere with the namespaces used by other OMG specifications or commonly used packages.		prefix `org.omg.dds`. The submitters are not aware of this package having been used by any other specification. *However*: This package has been used by the JacORB implementation of DDS in a non-standard fashion without the consent of the OMG. Therefore, implementations of the PSM defined here will not be able to coexist within the same Java class loader as the non-standard JacORB DDS implementation.
6.5.10	The "DDS for Lightweight CCM" specification introduced QoS profiles, a powerful capability for specifying QoS policy values in XML documents. Proposals shall include an API to allow DDS applications to use these profiles.	**Section 3, 8.2.5.3**	**Satisfied**: The PSM defines methods to create new entities and to set their QoS on based on the contents of QoS profiles.
6.5.11	In the event that the "Extensible and Dynamic Topic Types for DDS" specification is adopted before final submissions to this RFP are due, those submissions shall encompass any APIs introduced by that specification.	**Section 8.7**	**Satisfied**.
6.5.12	Proposals shall express the PSM in the form of Java packages containing classes and/or interfaces.	**Section 3**	**Satisfied**: This submission includes both a prose description of the PSM—this document—as well as a set of Java source files. Both are normative.

The proposed specification addresses the following optional requirements:

Table 2. Optional RFP Requirements

Requirement	Description	Reference	Remarks
6.6.1	Proposals may define how QoS may be configured and status notifications monitored using the Java Management	**—**	**Unsatisfied.**

	Extensions (JMX), a specification commonly used by monitoring and management tools and infrastructure.		
6.6.2	Proposals may define how data written and read through DDS may integrate with the built-in data serialization support in the Java platform (*e.g.* the interfaces `java.io.Serializable`, `java.io.ObjectInput`, and `java.io.ObjectOutput`).	**Section 9**	**Satisfied** by the Java Type Representation.
6.6.3	Proposals may define how dynamic applications can take advantage of the reflective capabilities of the Java platform. Possible uses of reflection include, but are not limited to:	*See below:*	
6.6.3.1	Proposals may comply with Java Beans-compatible design patterns.	**Section 8**	**Partially satisfied**: DDS types cannot be construc in a typical Java Bean fashion: they are not concre classes with default constructors. However, most accessor and mutator methods do form Java Bean properties, and listener callbacks follow some Jav Bean conventions.
6.6.3.2	Proposals may integrate the Dynamic Type and/or Dynamic Data APIs (to be defined by an anticipated "Extensible and Dynamic Topic Types for DDS" specification) with the built-in Java type introspection capability.	**Section 8.7.1.3**	**Satisfied** by the method `DynamicTypeFactory.createType(Clas` which creates a DDS `DynamicType` object base a Java Class object.
6.6.3.3	Proposals may provide reflective instantiation of, and/or access to, `DataReaders` and `DataWriters` corresponding to a given Java type.	**Sections 8.5.1, 8.6.1**	**Satisfied**. Types can be registered, and topics, reac and writers created, based on Java Class objects. Furthermore, the `Publisher` and `Subscriber` interfaces provide look-up methods that act on ger `Topic` objects as well as the name-based look-up methods defined in the DDS PIM.

6.6.4	Proposals may define how applications can load and access a DDS implementation compliant with the proposed PSM in the context of OSGi.	—	**Unsatisfied**. However, this scoping decision with respect to this specification shall not be taken to preclude the use of implementations with OSGi.

1.10 Responses to RFP Issues to Be Discussed

The specification discusses the following issues:

Table 3. RFP Issues to Be Discussed

Issue	Description	Reference	Remarks
6.7.1	Proposals shall discuss how performance is affected by the constructs and containers introduced by the new PSM and identify any situations that may degrade performance relative to the existing IDL PSM. As noted above, maintaining and improving performance is a significant goal of a new PSM.	**Section 1.7**	**Satisfied**. This PSM has no known performance degradation relative to the IDL PSM—and several performance improvements. Note that DDS performance with respect to types instantiated reflectively according to the Java Type Representation may not reach the same level as when types are backed by generated code. This limitation is intrinsic to the characteristics of reflective versus compiled code.
6.7.2	Proposals shall discuss the memory resources utilized by the constructs in the new PSM.	**Section 8.1.3**	**Satisfied**.
6.7.3	Proposals shall discuss how vendors will be able to expose APIs that are specific to their implementations. They shall discuss how vendors can make their extensions consistent with the style and design patterns of the specified PSM without affecting compatibility across implementations of it for applications that choose not to use those extensions.	**Section 8.1.6**	**Satisfied**: Vendors can provide extension QoS policies, which applications can look up and use reflectively without compile-time knowledge. In addition, vendors may define their own subtypes of standard types that expose additional capabilities; applications can access these either reflectively or using a type cast.

6.7.4	Proposals shall discuss the tradeoff between stylistic consistency with the IDL-based PSM and adherence to the coding conventions of Java.	**Section 1.7**	**Satisfied**. This PSM respects the functionality and structure of the PIM and places a strong emphasis on conventional Java practice for ease of use. However, it does draw on the IDL PSM in cases where it does not conflict with these goals and/or where it offers a performance benefit.
6.7.5	Proposals shall discuss how the PSM may be affected by language and standard library features that may appear in subsequent versions of the Java language such as Java 6 (already available) and Java 7 (as of this writing, not yet available).	—	**Satisfied**: nothing to discuss. This PSM would not be affected by any of the changes available in Java 6 SE. And as of this writing, Java SE 7 remains without an official JSR defining its scope.
6.7.6	Proposals shall discuss the memory and lifecycle management for DDS Entities.	**Section 8.1.3**	**Satisfied**.
6.7.7	Proposals shall discuss how interoperability can be assured between users of the new PSM and the IDL-based PSM.	—	**Satisfied**: nothing to discuss. This PSM has no bearing or impact on interoperability.
6.7.8	Proposals shall discuss how they allow applications to choose an implementation of the PSM at runtime and discuss why they chose the mechanism they did.	**Section 8.2.1**	**Satisfied** by the `Context` class, which is designed to resemble the `java.naming.InitialContext` class.
6.7.9	Proposals shall discuss why they made the choice of Java Edition(s) (*i.e.*, Java Micro Edition, Java Standard Edition, and Java Enterprise Edition) they did and what the expected impact is on the adoption of DDS technology among targeted Java users.	**Section 7.2**	**Satisfied**.

PART II – Java 5 Language PSM for DDS

2 Scope

This specification defines a platform-specific model (PSM) for the OMG Data Distribution Service for Real-Time Systems (DDS). It specifies an API only for the Data-Centric Publish-Subscribe (DCPS) portion of that specification; it does not address the Data Local Reconstruction Layer (DLRL). In addition, it encompasses (*a*) the DDS APIs introduced by [DDS-XTypes] and (*b*) an API to specifying QoS libraries and profiles such as were specified by [DDS-CCM].

This specification also defines a means of publishing and subscribing Java objects with DDS—the Java Type Representation—without first describing the types of those objects in another language, such as XML or OMG IDL.

3 Conformance

This specification consists of this document as well as a Java jar library file and the source files that generated it, identified on the cover page. All are normative. In the event of a conflict between them, the latter shall prevail.

Conformance to this specification parallels conformance to the DDS specification itself and consists of the same conformance levels. For example, an implementation may conform to the DDS Minimum Profile with respect to this PSM, meaning that all of the programming interfaces identified by the DDS specification as pertaining to that conformance level must be implemented in this PSM. The one exception to this rule is the Object Model Profile, which includes in part the Data Local Reconstruction Layer (DLRL); DLRL is outside of the scope of this PSM.

In addition to the conformance levels defined in the DDS specification itself, this PSM recognizes and implements the Extensible and Dynamic Types conformance level for DDS defined by the *Extensible and Dynamic Topic Types for DDS* specification [DDS-XTypes].

This PSM furthermore defines methods to create Entities and to set their QoS based on the XML QoS libraries and profiles defined by the *DDS for Lightweight CCM* specification. Implementations that support these XML QoS profiles shall implement these operations fully; other implementations shall throw `java.lang.UnsupportedOperationException`.

Finally, any conformant implementation must support at least one of the OMG-specified Type Representations defined by [DDS-XTypes] and/or in the Java Type Representation section of this specification (9 below).

4 References

4.1 Normative References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- **[DDS]** *Data Distribution Service for Real-Time Systems Specification*, version 1.2 (OMG document formal/2007-01-01).
- **[DDS-CCM]** *DDS for Lightweight CCM*, version 1.0 Beta 1 (OMG document ptc/2009-02-02).
- **[DDS-XTypes]** *Extensible and Dynamic Topic Types for DDS*, version 1.0 Beta 1 (OMG document ptc/2010-05-12).
- **[Java-Lang]** *The Java Language Specification, Third Edition*, published by Addison Wesley in 2005 with ISBN 0321246780
- **[XML]** *Extensible Markup Language* (XML), version 1.1, Second Edition (W3C recommendation, August 2006).

4.2 Non-Normative References

The following non-normative references are provided for informational purposes.

- **[JMS]** *Java Message Service Specification*, version 1.1 (Sun Microsystems, http://java.sun.com/products/jms/docs.html).

5 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

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.

Data Local Reconstruction Layer

The optional portion of the DDS specification used to provide the functionality required for an application for direct access to data exchanged at the DCPS layer. This later builds upon the DCPS layer.

Java Archive (JAR)

A zip file, whose name ends in the suffix `.jar`, that contains the compiled Java class files and

other artifacts that comprise a Java library.

Java Runtime Environment (JRE)

The environment within which Java applications execute. The JRE consists of an executing instance of a JVM, a set of class libraries, and potentially other components.

Java Virtual Machine (JVM)

An abstract computing machine capable of executing interpreted and/or compiled Java byte code. JVM implementations typically take the form of executables that run as processes under operating systems, but this style of implementation is not mandatory.

Platform-Independent Model (PIM)

An abstract definition of a facility, often expressed with the aid of formal or semi-formal modeling languages such as OMG UML, that does not depend on any particular implementation technology.

Platform-Specific Model (PSM)

A concrete definition of a facility, typically based on a corresponding PIM, in which all implementation-specific dependencies have been resolved.

6 Symbols

This specification does not define any symbols or abbreviations.

7 Additional Information

7.1 Changes to Adopted OMG Specifications

This specification does not extend or modify any existing OMG specifications.

7.2 Relationships to Non-OMG Specifications

This specification depends on version 5 of the Java Standard Edition platform. Service implementations may impose additional constraints; the nature and scope of these, if any, are unspecified.

Design Rationale (non-normative)

As of the publication of this specification, Java SE remains the predominant platform for the development and deployment of DDS Java applications.

- Introducing a dependency on Java EE would have brought little additional capability to the PSM and would have put it outside of the reach of many potential users, especially those deploying to embedded operating systems, many of which do support Java EE.
- Allowing the PSM to support Java ME would have made it less usable for the majority of

potential users: as of the publication of this specification, Java ME platforms lack support for many modern collections and Java language features. At the same time, support for Java ME would not have significantly increased the availability of implementations of this specification: many embedded platforms already support Java SE, and existing DDS vendors have not observed significant customer demand for Java ME support in existing products.

7.3 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc. (RTI)
- PrismTech

8 Java 5 Language PSM for DDS

The specification below is organized according to the module defined by the DDS specification and the types and operations defined within them.

8.1 General Concerns and Conventions

This section defines those elements of this specification that cut across multiple DDS modules.

8.1.1 Packages and Type Organization

This PSM is defined in a set of Java packages, the names of each beginning with the prefix `org.omg.dds`. Each of these contains a Java interface or abstract class for each type in the corresponding DDS module.

All of these packages, and the types within them, are packaged into a single JAR file, `omgdds.jar` (see *Annex A: Java JAR Library File*).

All those types that are abstract—including interfaces and abstract classes—are intended to be implemented concretely by the Service implementation. In addition, the subtypes defined by the implementation may expose additional implementation-specific properties and operations; however, the nature of these, if any, is undefined.

Design Rationale (non-normative)

This PSM divides the types it defines into multiple packages, rather than collocating them in a single package, for the following reasons:

- DDS defines a large number of types. Grouping them into multiple packages makes it clear which are more closely related to one another.
- The package organization improves traceability to the DDS PIM ([DDS]).
- The package organization parallels the namespace organization of the C++ PSM for DDS, facilitating cross-training across languages.

8.1.2 Implementation Coexistence

To facilitate the coexistence of multiple DDS implementations within the same JVM instance, each implementation of this PSM shall cooperate at the API level with other JVM-local implementations in at least the following ways:

- It shall be possible to pass an instance of any value type (see section 8.2.3) created by one DDS implementation to a method implemented by another. For example, the method `DataWriter.write` optionally accepts an argument of type `InstanceHandle`; this object may have been created by the same DDS implementation that created the `DataWriter` or by another DDS implementation.
- It shall be possible to read or take samples from a `DataReader` provided by one DDS implementation and immediately write them using a `DataWriter` provided by another DDS implementation, provided that the samples are of a DDS type compatible with that `DataWriter`.

Note that passing an object from one implementation to another may incur a performance cost, as the "receiving" implementation may have to copy the object in question before operating on it.

Otherwise, unless elsewhere noted in this specification, a Service implementation may raise an exception or behave in an undefined way if it encounters a concrete type defined by another party. For example, the concrete `WaitSet` implementation provided by one DDS vendor need not support the attachment of `Condition` implementations provided by another DDS vendor.

8.1.3 Resource Management

The use of interfaces instead of classes requires the introduction of an explicit factory pattern for the construction of objects of all DDS types. For some types (Entities in particular), this pattern is already explicit in the DDS PIM. For other types (such as QoS policies), it is a property solely of the PIM-to-PSM mapping. These latter types—those without PIM-defined factory construction methods—serve as their own factories. Each is represented as an abstract class with one or more static factory methods. These methods are named according to the convention `new<`*`ClassName`*`>` in order to resemble constructor invocations and are amenable to use with the Java 5 static import facility.

This PSM maps the factory deletion methods of the DDS PIM (*e.g.* `DomainParticipant.delete_publisher`) to `close` methods on the "product" interfaces themselves (*e.g.* `Publisher.close`). Closing an Entity implicitly closes all of its contained objects, if any. For example, closing a `Publisher` also closes all of its contained `DataWriters`.

Design Rationale (non-normative)

The `close` destruction design pattern is intended to be familiar to those developers who have used `java.io` stream APIs and/or [JMS] and eliminates the possibility that an object could be deleted using a factory other than the one that created it.

Users of this PSM are recommended to call `close` once they are finished using such

heavyweight objects. In addition, implementations *may automatically* close objects that the JRE deems to be no longer in use—for example, they may call `close()` in an `Object.finalize()` override—subject to the following restrictions:

- Any object to which the application has a direct reference is still in use.
- Any entity with a non-null listener is still in use.
- Any object that has been explicitly retained is still in use
- The creator of any object that is still in use is itself still in use.

8.1.4 Concurrency and Reentrancy

It is expected that most Service implementations will be used frequently in multi-threaded environments. Therefore, for the sake of portability, this PSM constrains the level of thread safety that applications may expect:

- All `DataReader` and `DataWriter` operations shall be reentrant.
- All `Topic` (and other `TopicDescription` extension interfaces), `Publisher`, `Subscriber`, and `DomainParticipant` operations shall be reentrant with the exception that `close` may not be called on a given object concurrently with any other call of any method on that object or on any contained object.
- All `Bootstrap` and `DomainParticipantFactory` operations shall be reentrant with the exception that `DomainParticipantFactory.close` may not be called on a given object concurrently with any other call of any method on that object or on any contained object.
- All `WaitSet` and `Condition` (including `Condition` extension interfaces) operations shall be reentrant with the exception that their `close` methods may not be called on a given object concurrently with any other call of any method on that object.
- Code within a DDS listener callback may not safely call any method on any DDS Entity but the one on which the status change occurred.
- Any method of any value type may be non-reentrant.

A vendor may choose to provide stronger guarantees than the rules above, but if so, those guarantees are unspecified.

Design Rationale (non-normative)

Objects that are likely to "own" mutexes within their implementation need not permit `close` invocations currently with other method invocations. This is to allow implementations to dispose of these mutexes within the `close` method without creating a race condition or requiring an additional level of locking.

Method invocations are restricted within listener callbacks in order to avoid deadlocks, especially in Service implementations that invoke callbacks within Service-managed threads.

8.1.5 Method Signature Conventions

This PSM maps the underscore-formatted names of the DDS PIM and IDL PSM (such as `get_qos`) into conventional Java "camel-case" names (such as `getQos`). This mapping makes the API look more familiar to Java developers and makes it interoperate better with Java reflective technologies that expect this naming convention.

Properties defined by the DDS PIM are expressed as sets of accessor and mutator methods. The signatures of these methods conform to the following convention:

- Mutators are named `set<`*`PropertyName`*`>`. (For example, the mutator for a property "Foo" would be named `setFoo`.) They take a single argument—the new value of the property—and return the enclosing object in order to facilitate method chaining.
- Accessors for properties that are either of unmodifiable objects (such as those of primitive types, primitive box types, or strings) or pointers to the internal state of an object are named `get<`*`PropertyName`*`>`. (For example, the accessor for an integer property "Foo" would be named `getFoo`.) They take no arguments.
- Accessors for properties that are of mutable types, and that may change asynchronously after they are retrieved, are named `get<`*`PropertyName`*`>`. They take a pre-allocated object of the property type as their first argument, the contents of which shall be overwritten by the method. To facilitate method chaining, these methods also return a reference to this argument. This pattern forces the caller to make a copy, thereby avoiding unexpected changes to the property. An Entity's status is an example of a property of this kind.

8.1.6 API Extensibility

Implementation-specific extensions to the types specified by this PSM are by definition unspecified. However, implementations may provide such a capability by providing extended implementation-specific interfaces and returning instances of these interfaces from the specified factory methods.

Implementations shall not place their extensions, if any, in any interface or class in the package `org.omg.dds` or in any other package whose name begins with that prefix.

8.2 Infrastructure Module

This PSM realizes the Infrastructure Module from the DDS specification with two packages: `org.omg.dds.core` and `org.omg.dds.core.policy`. The latter contains all QoS policy classes, since a given QoS policy may apply to multiple DDS Entity types. The former contains all other Infrastructure types, including for example `Entity` and `Condition` base interfaces.

Design Rationale (non-normative)

These two packages have been made distinct from one another for two reasons: First, the QoS policies constitute a significant proportion of the total set of types in the Infrastructure Module,

and the contents of the module are thus easier to understand when they are divided along this line. Second, a dedicated package for QoS policies makes the code completion features of modern programming environments easier to use, because it allows users to narrow the set of classes through which they must search in order to find the one they're looking for.

The term "core" has been preferred to "infrastructure" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "core" as well.

8.2.1 `Bootstrap` Class

A `Bootstrap` object represents an instantiation of a Service implementation within a JVM. It is the "root" for all other DDS objects and assists in their creation by means of an internal service-provider interface. All stateful types in this PSM implement an interface `DDSObject`, through a `getBootstrap` method on which they can provide access to the `Bootstrap` from which they are ultimately derived. (`Bootstrap` itself implements this interface; a `Bootstrap` always returns `this` from its `getBootstrap` operation.)

The `Bootstrap` class allows implementations to avoid the presence of static state, if desired. It also allows multiple DDS implementations—or multiple versions of the "same" implementation—to potentially coexist within the same Java run-time environment. A DDS application's first step is to instantiate a `Bootstrap`, which represents the DDS implementation that it will use. From there, it can create all of its additional DDS objects.

The `Bootstrap` class is abstract. To avoid compile-time dependencies on concrete `Bootstrap` implementations, an application can instantiate a `Bootstrap` by means of a static `createInstance` method on the `Bootstrap` class. This method looks up a concrete `Bootstrap` subclass using a Java system property containing the name of that subclass. This subclass must be provided by implementers and will therefore have an implementation-specific name.

8.2.2 Error Handling and Exceptions

The PSM maps the `ReturnCode_t` type from the DDS PIM into a combination of standard Java exceptions (where their semantics match those expressed in the PIM) and new exception classes defined by this PSM. This mapping is as follows:

- With the exception of `java.util.concurrent.TimeoutException`, all exceptions are unchecked (that is, they extend `java.lang.RuntimeException` directly or indirectly).
- The exception classes defined by this PSM extend the base class `DDSException`. All of the PSM-defined exception classes are defined in the package `org.omg.dds.core`. All of these classes are abstract so as not to specify the representation of state; implementations shall provide concrete implementations.

Table 4 `ReturnCode_t` → exception mapping

`ReturnCode_t` Value	Exception Class
`RETCODE_OK`	Normal return; no exception
`RETCODE_NO_DATA`	An informational state (e.g. a Boolean result) attached to a normal return; no exception
`RETCODE_ERROR`	`DDSException`
`RETCODE_BAD_PARAMETER`	`java.lang.IllegalArgumentException`
`RETCODE_TIMEOUT`	`java.util.concurrent.TimeoutException`
`RETCODE_UNSUPPORTED`	`java.lang.UnsupportedOperationException`
`RETCODE_ALREADY_DELETED`	`AlreadyClosedException`
`RETCODE_ILLEGAL_OPERATION`	`IllegalOperationException`
`RETCODE_NOT_ENABLED`	`NotEnabledException`
`RETCODE_PRECONDITION_NOT_MET`	`PreconditionNotMetException`
`RETCODE_IMMUTABLE_POLICY`	`ImmutablePolicyException`
`RETCODE_INCONSISTENT_POLICY`	`InconsistentPolicyException`
`RETCODE_OUT_OF_RESOURCES`	`OutOfResourcesException`

In addition, this PSM permits implementations to throw exceptions to indicate errors in operations that in the PIM return an object reference. The PIM uses the convention of modeling failure conditions as operation return results, making it impossible to provide finer failure-detection granularity than a simple nil/non-nil result check in the case of methods that must return something other than a return code. The Java language, with built-in exception support, eliminates that restriction, and this PSM takes advantage of that fact.

Design Rationale (non-normative)

This PSM uses checked and unchecked exceptions according to the following rationale: Where the exception represents a *fault*—a design flaw, implementation mistake, or runtime failure—it is unchecked. Where it represents a *contingency*—an uncommon-but-expected return scenario, for which the caller is expected to have a coping strategy—it is checked[2].

Most exceptions in the DDS API represent faults, not contingencies.

Within each category, this PSM reuses existing JRE exception classes when they are available and appropriate.

[2] The fault/contingency model of Java exceptions was first described by Barry Ruzek, then of BEA, in late 2006 or early 2007 in the article *Effective Java Exceptions*. This article was originally published at http://dev2dev.bea.com/pub/a/2006/11/effective-exceptions.html and is now available at http://crmondemand.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html.

8.2.3 Value Types

All DDS types with value semantics implement the interface `org.omg.dds.core.Value`[3]. These include QoS, QoS policy, status, time, and other types.

The `Value` interface extends the standard Java SE interfaces `java.lang.Cloneable` and `java.io.Serializable`, allowing objects of implementing types to be copied by value as well as serialized and deserialized using built-in Java mechanisms.

It also defines a small number of additional methods. It defines a method `copyFrom` that accepts a source object of the same type as the object itself. This method overwrites the state of the target object ("`this`") with the state of the argument object; it is similar to `clone` but does not require allocating a new object. `Value` implementers are also expected to override their inherited implementations of `Object.equals` and `Object.hashCode` in order to enforce value semantics.

Some value types come in modifiable and unmodifiable varieties—notably QoS and QoS policies. The "modifiable" interface extends the "unmodifiable" one.

- The latter provides an operation `modify` that returns an instance of the former. Classes that implement the unmodifiable interface but not the modifiable one shall implement this operation to return a new modifiable object containing a copy of the state of the target unmodifiable object. Classes that implement the modifiable interface shall return a pointer to themselves.
- Modifiable value types with unmodifiable counterparts have an inverse operation: `finishModification`. In many cases, calling this operation is optional, as modifiable interfaces extend unmodifiable ones. However, in some cases, a truly unmodifiable object is desirable, such as when it will be shared among threads without locking.

8.2.4 Time and Duration

This PSM maps the DDS `Time_t` and `Duration_t` types into the value types `Time` and `Duration` respectively. These classes can provide their magnitude using a variety of units (expressed using `java.util.concurrent.TimeUnit`).

Design Rationale (non-normative)

The names of these types omit the underscore and 't' characters from the ends of their names. That naming convention, while common among C POSIX programmers, is not conventional in Java.

[3] The term "value type" refers to any data type for which object identity is considered to be established solely based on the state of the objects of that type. Such types generally provide deep copy and comparison operations. (For example, integers are an example of a value type: every occurrence of the quantity 42 is considered to refer to the same number as every other.) The term should not be confused with an IDL `valuetype` as defined by the CORBA specification.

8.2.5 QoS and QoS Policies

QoS-related types fall into two categories, as expressed in the DDS PIM: individual QoS policies (such as reliability) and the collections of policies that apply to a particular DDS Entity type. This PSM represents the former with the base interface `org.omg.dds.core.policy.QosPolicy` and the latter with the base interface `org.omg.dds.core.Qos`.

8.2.5.1 QoS Policies

The DDS PIM represents each QoS policy in three ways; this PSM maps them as follows:

Table 5 QoS policy representation

DDS PIM	Java 5 PSM
QoS policy structure containing the state of an instance of that policy	QoS policy interface extending `org.omg.dds.core.policy.QosPolicy`. Each policy provides Java Bean-style properties.
Unique QoS policy ID, represented by an instance of the enumeration `QosPolicyId_t`	Unique QoS policy ID, represented by an instance of the nested abstract class `org.omg.dds.core.policy.QosPolicy.Id`. The numeric value given in the IDL PSM is preserved in the `Id` integer-valued method `getPolicyIdValue()`.
Unique QoS policy name, represented by a string property `QosPolicy.name`	Unique QoS policy ID, represented by an instance of the nested abstract class `org.omg.dds.core.policy.QosPolicy.Id`. The name is preserved in the `Id` string-valued method `getPolicyName()`.

8.2.5.2 Entity QoS

Each Entity QoS (e.g. `DataReaderQos`) is an interface extending `org.omg.dds.core.Qos`. These sub-interfaces provide direct access to their policies as in the IDL PSM. However, the base interface also provide for generic access using the `java.util.Map` interface. This interface allows applications to look up policies by ID and to iterate over them in a generic way, including vendor-specific extension policies, without introducing compile-time dependencies on vendor-specific APIs.

The contents of a QoS object are only meaningful in relation to the current QoS or default QoS of some Entity or group of Entities. Therefore, these objects cannot be created directly; they can only be cloned from pre-existing state maintained by the Service implementation.

QoS objects as returned by Entities shall be immutable; applications shall never observe them to change. Applications that wish to modify QoS values must first call `modify` to obtain a modifiable QoS object; after making their desired modifications, they must pass their new QoS values to `setQos`.

Design Rationale (non-normative)

The copy-on-write idiom described above has several benefits:

- The `getQos` operation can operate maximally efficiently: it need not allocate any memory or perform any copies.
- The immutable result of `getQos` can be used safely concurrently from multiple threads.
- The `getQos` and `setQos` methods form a conventional Java-Bean-style property.

8.2.5.3 QoS Libraries and Profiles

The *DDS for Lightweight CCM* specification [DDS-CCM] defines a format for QoS libraries and profiles. These libraries and profiles provide a mechanism for entity QoS configuration administration. This PSM provides the following APIs for accessing these administered QoS configurations:

- The `org.omg.dds.core.Entity` interface allows any Entity's QoS to be set based on the names of a QoS library and profile.
- Each Entity factory interface—`DomainParticipantFactory`, `DomainParticipant`, `Publisher`, and `Subscriber`—provides methods to create new "product" Entities and to set their default QoS based on the names of a QoS library and profile.

8.2.6 Entity Base Interfaces

As in the DDS PIM, all Entity interfaces extend—directly or indirectly—the interface Entity. In this PSM, this interface is generic; it is parameterized by the Entity's QoS and listener types. These parameters allow applications to call common operations like `getQos` or `getListener` in a type-safe way while still working with Entities polymorphically.

Also as in the DDS PIM, Entities other than `DomainParticipant` extend the interface `DomainEntity`. These Entities provide operations to get the creating parent Entity; in this PSM, this operation is the polymorphic `DomainEntity.getParent`.

8.2.7 Entity Status Changes

This section describes the objects pertaining to the status changes of DDS Entities: the Status types themselves, listeners, conditions, and wait sets.

8.2.7.1 Status Classes

This PSM represents each status identified by the DDS PIM as an abstract class extending `org.omg.dds.core.Status`, which in turn extends `java.util.EventObject`.

The DDS PIM also identifies statuses using a "status kind"; these are composed into a mask that is used when setting listeners and at other times. This PSM represents status kinds using the `java.lang.Class` instances of the corresponding status classes and status masks as

`java.util.Set`s of such status classes.

Status objects passed to listeners in callbacks may be pooled and reused by the implementation. Therefore, applications that wish to retain these objects—or any objects found within them, such as instance handles—for later use outside of the callback are responsible for copying them.

8.2.7.2 Listeners

This PSM maps the Listener interface from the DDS PIM to the empty marker interface `java.util.EventListener` interface defined by the Java SE standard library.

For each listener sub-interface (*e.g.* `DataWriterListener`), this PSM provides a concrete implementation of that interface in which all methods have empty implementations. These concrete classes are named like the listener interfaces they implement, but with the word "Listener" replaced by "Adapter."

In the DDS PIM, each listener callback receives two arguments: the Entity, the status of which has changed, and the new value of that status. In this PSM, the former is unnecessary and is omitted: it is available through the read-only `Source` property of the status object.

Design Rationale (non-normative)

The listener + adapter design pattern is consistent with that used in the standard AWT and Swing UI libraries and elsewhere. It allows applications that are only interested in a subset of the callbacks provided by an interface to override only those methods and ignore the others.

This PSM distinguishes between lower-level listener interfaces, the implementations of which are likely to do type-specific things, and higher-level listener interfaces, the implementations of which are likely to do type-agnostic things.

- The former category includes `TopicListener`, `DataReaderListener`, and `DataWriterListener`. These classes are generic; their type parameters match that of the Entities on which they are set. This convention allows applications to read and write data within the context of a callback in a statically type-safe way.
- The latter category includes `PublisherListener`, `SubscriberListener`, and `DomainParticipantListener`. The `Topic`s, `DataReader`s, and `DataWriter`s passed to these listeners' callbacks are parameterized with the generic wildcard '`?`'. Because of this difference between these listeners and those in the former category, there are no inheritance relationships between these categories, unlike in the PIM.

8.2.7.3 Conditions

Conditions extend the base interface `org.omg.dds.core.Condition`.

The interface `StatusCondition`, which extends `Condition`, is a generic interface with a type parameter that is the type of the Entity to which it belongs. This type parameter allows its `getEntity` method to be both polymorphic and type safe.

8.2.7.4 Wait Sets

Wait sets extend the base interface `org.omg.dds.core.WaitSet`.

In the DDS PIM, an application indicates its intention to wait for a condition to be triggered by invoking the operation `WaitSet.wait`. However, in Java, this operation overloads unintentionally with the inherited method `Object.wait`. This inherited method has a different meaning; the overload is inappropriate. Therefore, this PSM maps the DDS PIM `wait` operation to the more explicit method name `waitForConditions`.

8.3 Domain Module

This PSM realizes the Domain Module from the DDS specification with the package `org.omg.dds.domain`. This package contains `DomainParticipant`, `DomainParticipantFactory`, and so forth.

8.3.1 `DomainParticipantFactory` Interface

The `DomainParticipantFactory` is a per-`Bootstrap` singleton. An instance of this interface can be obtained by passing that `Bootstrap` to the factory's `getInstance` method.

8.3.2 `DomainParticipant` Interface

This PSM represents the `DomainParticipant` classifier from the DDS PIM with the interface `org.omg.dds.domain.DomainParticipant`.

8.4 Topic Module

This PSM realizes the Topic Module from the DDS specification with the packages `org.omg.dds.type` and `org.omg.dds.topic`.

8.4.1 Type Support

As in the DDS PIM, each type to be published or subscribed with DDS is represented by a class extending `org.omg.dds.type.TypeSupport`. Applications obtain instances of these interfaces by calling the static base class operation `newTypeSupport`, passing this method the Java `Class` object of the type they wish to support and optionally a name under which that type should be registered. If no such name is provided, the type shall be registered under the fully qualified name of the provided `Class`.

This PSM modifies slightly the capability for type registration provided by the DDS PIM. In the PIM, types are registered by invoking a `TypeSupport.register_type` operation. Subsequently, applications provide the registered type name to the `DomainParticipant.create_topic` operation in order to refer to the registered type. This PSM instead asks applications to instantiate each `TypeSupport` object with a name and then provide that `TypeSupport` itself to the `create_topic` method.

Design Rationale (non-normative)

By requiring the application to pass an instance of the generic `TypeSupport` interface to the `createTopic` method, this PSM maintains unbroken static type safety all the way from type registration to data publication or reception. A pattern of type access based on the name strings would require a type cast.

8.4.2 `Topic` Interface

The `Topic` interface adds only a single operation to the set of those it inherits from its `TopicDescription` and `DomainEntity` super-types: an accessor for the inconsistent topic status. `Topic`—like all `TopicDescriptions`, and like `DataReader` and `DataWriter`—is a generic interface with a type parameter that identifies the type of the data with which it is associated. Although `Topic` provides no type-specific operations, its type parameter preserves type safety from `Topic` creation (actually all the way from type registration) through data publication and/or subscription.

8.4.3 `ContentFilteredTopic` and `MultiTopic` Interfaces

`ContentFilteredTopic` and `MultiTopic` are generic interfaces with type parameters that identify the types of the data with which they are associated.

Note that the type parameter of a `ContentFilteredTopic` does not need to match that of its related `Topic` exactly; it can be any supertype. So for example, if the user-defined type `Bar` extends the user-defined type `Foo`, a `ContentFilteredTopic<Foo>` can wrap a `Topic<Bar>`.

8.4.4 Discovery Interfaces

The data types pertaining to the DDS built-in discovery topics are contained in the package `org.omg.dds.topic` as well. These types provide only accessors for their state, not mutators, to reflect the read-only (from an application's point of view) nature of discovery.

8.5 Publication Module

This PSM realizes the Publication Module from the DDS specification with the package `org.omg.dds.pub`.

Design Rationale (non-normative)

The term "pub" has been preferred to the longer "publication" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "pub" as well.

8.5.1 `Publisher` Interface

`Publishers` are represented by instances of the `org.omg.dds.pub.Publisher` interface.

In addition to the methods defined for this interface by [DDS], it additionally provides a `lookupDataWriter` overload that acts on the basis of a `Topic` object rather than solely on the topic's name. This overload is provided for the sake of additional static type safety.

8.5.2 `DataWriter` Interface

`DataWriter`s are represented by instances of the `org.omg.dds.pub.DataWriter` interface. This is a generic interface, parameterized by the type of the data samples to be written by a given writer. The DDS PIM distinguishes between a type-specific `DataWriter` (`FooDataWriter`) and one whose type is not statically known (`DataWriter` itself); these are related by an inheritance relationship. This PSM makes no such distinction: Java's generic wildcard syntax (`DataWriter<?>`) makes it possible to express all type-specific `DataWriter` operations on the `DataWriter` interface itself; there is no `FooDataWriter`.

For most type-specific operations, the DDS PIM provides variants that accept an explicit timestamp (to allow applications to manage the passage of time themselves) and variants that do not (indicating that the Service implementation should provide this); these two sets of operations use different naming conventions. In addition, the PIM includes an instance handle parameter in the signatures of these operations, despite the fact that not all types are keyed and therefore have any use for instance handles. These design choices reflect the existence of the IDL PSM: IDL does not support method overloading. Java does; therefore, the provision of timestamps and/or instance handles is optional and is handled by means of method overloads. For example, the `write` method provides the following overloads: one accepting a data sample only, another accepting a sample and an instance handle, and another accepting both of these as well as a timestamp. Users of `DataWriter`s of unkeyed types may choose to call the overloads that accept instance handle arguments; if they do, the handle argument must be a nil handle (as explained in the DDS PIM).

8.6 Subscription Module

This PSM realizes the Subscription Module from the DDS specification with the package `org.omg.dds.sub`.

Design Rationale (non-normative)

The term "sub" has been preferred to the longer "subscription" for the sake of brevity (such as when using fully qualified names) and for consistency with the C++ PSM for DDS, which uses the term "sub" as well.

8.6.1 `Subscriber` Interface

`Subscriber`s are represented by instances of the `org.omg.dds.sub.Subscriber` interface.

In addition to the methods defined for this interface by [DDS], it additionally provides a `lookupDataReader` overload that acts on the basis of a `TopicDescription` object rather than solely on the topic description's name. This overload is provided for the sake of additional static type safety.

8.6.2 `Sample` Interface

This PSM follows the guidance of the DDS PIM rather than of the IDL PSM: it represents data samples as single objects that incorporate both data and metadata. Each sample is represented by an instance of the `org.omg.dds.sub.Sample` interface. It provides its data via a `getData` method; if there is no valid data (corresponding to a false value for `SampleInfo.valid_data` in the IDL PSM), this operation returns `null`. It provides its metadata (corresponding to the other `SampleInfo` properties in the IDL PSM) as read-only Java-Bean-style properties.

The `Sample` interface also defines a nested interface: `Sample.Iterator`, an iterator that extends `java.util.ListIterator`. An iterator of this type provides read-only access to an ordered series of samples of a single type; such iterators are used by the `DataReader` `read` and `take` methods (see below).

8.6.3 `DataReader` Interface

`DataReader`s are represented by instances of the `org.omg.dds.sub.DataReader` interface. This is a generic interface, parameterized by the type of the data samples to be read by a given reader. The DDS PIM distinguishes between a type-specific `DataReader` (`FooDataReader`) and one whose type is not statically known (`DataReader` itself); these are related by an inheritance relationship. This PSM makes no such distinction: Java's generic wildcard syntax (`DataReader<?>`) makes it possible to express all type-specific `DataReader` operations on the `DataReader` interface itself; there is no `FooDataReader`.

The `DataReader` interface provides an extensive set of `read` and `take` method overloads. In addition to the distinction between read vs. take semantics (as defined in the DDS PIM), these operations come in two "flavors": one that loans samples from a Service pool and returns a `Sample.Iterator` and another that deeply copies into an application-provided `java.util.List`.

- Applications that read or take loans must eventually return those loans; this PSM maps the `return_loan` operation from the DDS PIM to an operation `returnLoan` on the `Sample.Iterator`.
- Applications that read or take copies may provide to the Service destination `List`s with any number of `Sample`s already in them (including empty `List`s). Regardless of the number of `Sample`s already in the list when the method is called, when it returns, the `List` shall contain the number of `Sample`s requested by the application (or fewer, if fewer were available). The Service implementation may—for example, in order to avoid object allocations—elect to overwrite the contents of any `Sample`s that are passed into it by invocations of these methods.

The `read` and `take` operations defined by the DDS PIM do not take advantage of overloading, because they were designed with the IDL PSM in mind, and IDL does not support overloading. Java does; therefore, this PSM both simplifies the operations' signatures as well as captures commonalities among them as follows:

- Several operation variants accept large numbers of infrequently used parameters (for example, sets of sample, instance, and view states). These operations have been split into two overloaded methods: one that accepts the minimum number of arguments and a second that accepts the full list.
- Operations accepting `ReadConditions` in the PIM have names ending in "`_w_condition`." This PSM removes this suffix, transforming these operations into overloads.
- Operations accepting instance handles in the PIM have "`_instance`" in their names. This PSM removes this infix, transforming these operations into overloads.
- This PSM renames both of the operation families `read_/take_next_sample` and `read_/take_next_instance` to simply `read/takeNext`, transforming these operations into overloads of one another.

8.7 Extensible and Dynamic Topic Types Module

This section of this specification addresses those additions to DDS introduced by the *Extensible and Dynamic Topic Types for DDS* specification [DDS-XTypes]. The additions fall into the following categories:

- **Types pertaining to `TypeObject` Type Representations** are defined in the package `org.omg.dds.type.typeobject`.
- **Types pertaining to the Dynamic Language Binding** are defined in the package `org.omg.dds.type.dynamic`.
- **The `TypeKind` enumeration**, which pertains to both of the above, is defined in the package `org.omg.dds.type`.
- **The built-in types** are defined in the package `org.omg.dds.type.builtin`.
- **Extensions by [DDS-XTypes] to types defined by [DDS]** (such as the built-in topic data types) are contained within those types.

8.7.1 Dynamic Language Binding

The Dynamic Language Binding, as defined by [DDS-XTypes], consists of `DynamicType`, `DynamicTypeMember`, `DynamicData`, their respective factories, and several "descriptor" value types.

8.7.1.1 `DynamicTypeFactory` and `DynamicDataFactory` Interfaces

These abstract factories are per-`Bootstrap` singletons. The static `delete_instance` operations defined in [DDS-XTypes] have been omitted in this PSM; the Service shall manage the life cycles of the factories.

8.7.1.2 `DynamicTypeSupport` Interface

The interface `DynamicTypeSupport` defined by [DDS-XTypes] does not provide any capability beyond what the generic `TypeSupport` interface provided by this PSM already provides. Therefore, it has been omitted from this PSM.

8.7.1.3 `DynamicType` and `DynamicTypeMember` Interfaces

These interfaces are expressed in this PSM according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Operations that provide their result as an in-out value in their first parameter and return `DDS::ReturnCode_t` have been changed such that they instead return their results directly. (This change, made for the convenience of the caller, is possible because `DDS::ReturnCode_t` is mapped to a set of exceptions in this PSM.)
- The `equals` and `clone` operations on these types have been mapped to overrides of the Java-standard `Object.equals` and `Object.clone`, respectively.
- `DynamicTypeMember` is a reference type, and instances of it are obtained from `DynamicType.addMember`. This change avoids the need to provide an additional factory method for `DynamicTypeMember` instances.
- On each type, the operations `get_annotation_count` and `get_annotation` (by index) have been unified into a single `getAnnotations` method that returns a list of annotations. The lists returned from these methods shall not be modifiable.

In addition to the methods specified by [DDS-XTypes], `DynamicTypeFactory` provides one additional factory method: `createType(Class<?>)`. This method shall inspect the given type reflectively in accordance with the Java Type Representation (section 9 below) and instantiate an equivalent `DynamicType` object.

8.7.1.4 `DynamicData` Interface

This interface is expressed in this PSM according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Operations that provide their result as an in-out value in their first parameter and return `DDS::ReturnCode_t` have been changed such that they instead return their results directly. (This change, made for the convenience of the caller, is possible because `DDS::ReturnCode_t` is mapped to a set of exceptions in this PSM.)
- The `equals` and `clone` operations on these types have been mapped to overrides of the Java-standard `Object.equals` and `Object.clone`, respectively.
- Methods dealing with unsigned integer types have been omitted. Applications may access unsigned data using the signed type of the same size (*e.g.* `UInt32` becomes `Int32`), which preserves bitwise representation but not logical value, or by using the signed type one size up (*e.g.* `UInt32` becomes `Int64`), which preserves logical value but not representation (and may therefore require additional range checking by the

implementation). In the case of UInt64, the “type one size up” is `java.math.BigInteger`.

- The 128-bit `Float128` type has been represented using `java.math.BigDecimal`.

8.7.1.5 Descriptor Interfaces

The following interfaces are values types with modifiable and unmodifiable variants, as described in section 8.2.3 above:

- `AnnotationDescriptor` (and `ModifiableAnnotationDescriptor`)
- `MemberDescriptor` (and `ModifiablememberDescriptor`)
- `TypeDescriptor` (and `ModifiableTypeDescriptor`)

8.7.2 Built-in Types

[DDS-XTypes] specifies four built-in types: `DDS::String`, `DDS::Bytes`, `DDS::KeyedString`, and `DDS::KeyedBytes`.

- `DDS::String` is mapped to `java.lang.String`.
- `DDS::Bytes` is mapped to `byte[]`.
- `DDS::KeyedString` and `DDS::KeyedBytes` are mapped to modifiable value type interfaces.

The `DataReader` and `DataWriter` specializations for these built-in types provide additional overloaded methods not implied by the generic versions of these interfaces. Therefore, this PSM defines extended interfaces `StringDataReader`, `StringDataWriter`, `BytesDataReader`, `BytesDataWriter`, and so on. It furthermore provides additional `Subscriber.createDataReader` and `Publisher.createDataWriter` variants specially tailored to the built-in types that return these extended interface types to allow applications to take advantage of these additional methods while maintaining static type safety. Note that the existence of these built-in-type-specific `Publisher` and `Subscriber` factory methods does not imply that the generic versions of these methods do not apply to the built-in types; they do.

8.7.3 Representing Types with `TypeObject`

The types in this package are expressed as modifiable value types according to the mapping rules expressed elsewhere in this document. In addition, the following changes to this mapping have been made:

- Top-level constants are moved into related interfaces, for example: `Member.MEMBER_ID_INVALID`.
- Enumerations of member ID values are nested final classes within the interfaces for which they provide the member’s IDs. These classes have constant integer fields, for example: `MapType.MemberId. BOUND_MAPTYPE_MEMBER_ID`.

9 Java Type Representation and Language Binding

The Java Type Representation defined in this section provides a means for Java developers to publish and subscribe to DDS topics typed by plain Java objects without resorting to code generation or the reflective style of the Dynamic Language Binding.

By its very nature as an expression of the Java programming language, this Type Representation implicitly and simultaneously defines a Language Binding for DDS types. That is, a Java type necessarily defines a Java API to itself as part of its definition. Therefore, this Type Representation is intended for the run-time use of implementations of this PSM. While this specification does not preclude Service implementations from using this Type Representation for other purposes—for example, generating a Plain Language Binding in C for a DDS type represented in Java—such uses are non-normative and unspecified.

The Java platform provides a mechanism by which Java type definitions can be used to define how objects can be serialized for transmission over a network: the `java.io.Serializable` interface and its related types. Since the transmission of data from Java programs over DDS is a related problem, this specification builds on that mechanism. Any Java type that implements `Serializable` (directly or indirectly) shall be available for publishing and/or subscribing over DDS as defined below. *Note* that the DDS serialization of a type will not generally be the same as the JRE serialization of the same type, even if the type designer's specification of which data to serialize can be shared between these two mechanisms.

9.1 Default Mappings

The following table defines the default mappings from Java type system definitions to DDS type system ones:

Table 6 — Default type mappings

Java Type	DDS Type
`int`, `java.lang.Integer`	`Int32`
`short`, `java.lang.Short`	`Int16`
`long`, `java.lang.Long`	`Int64`
`float`, `java.lang.Float`	`Float32`
`double`, `java.lang.Double`	`Float64`
`char`, `java.lang.Character`	`Char8`
`byte`, `java.lang.Byte`	`Byte`
`boolean`, `java.lang.Boolean`	`Boolean`
`java.lang.String`	`string<Char8>`
`java.util.Map`	`map`

java.lang.Collection, array	sequence
java.lang.Object	Structure

A type designer may modify these defaults on a type-by-type and/or field-by-field basis by applying the annotation org.omg.dds.type.SerializeAs:

```
public @interface SerializeAs {
   public TypeKind value();
   …
}
```

9.2 Metadata

The type system metadata represented with built-in annotations in the IDL Type Representation (such as @Key, @ID) shall be represented by equivalent Java annotations unless otherwise noted. These annotations are in the package org.omg.dds.type.

The annotations in this package logically govern the behavior of concrete classes, not of polymorphic interfaces. As such, they may be applied to classes or to their fields, as appropriate. Interface designers wishing to document the DDS serialization of a type may additionally apply them to interfaces or to property accessor and/or mutator methods; however, they have no specified behavior in such cases.

9.3 Primitive Types

By default, Java primitive types are mapped to DDS primitive types as defined in Table 6 above. The @SerializeAs annotation may be used to modify these mappings as follows:

Table 7 — Customized primitive type mappings

DDS Type	Permitted Java Primitive Types
Int32	int, java.lang.Integer
UInt32	int, long, java.lang.Integer, java.lang.Long
Int16	short, java.lang.Short
UInt16	short, int, java.lang.Short, java.lang.Integer
Int64	long, java.lang.Long
UInt64	long, java.lang.Long, java.math.BigInteger
Float32	float, java.lang.Float
Float64	double, java.lang.Double
Float128	double, java.lang.Double, java.math.BigDecimal

`Byte`	`byte`, `java.lang.Byte`
`Boolean`	`boolean`, `java.lang.Boolean`
`Char8`	`char`, `java.lang.Character`
`Char32`	`char`, `int`, `java.lang.Character`, `java.lang.Integer`

The DDS Type System ([DDS-XTypes]) defines unsigned integer types; the Java type system does not. As a result, this Type Representation must map unsigned values to "equivalent" signed types. Type designers have two choices, reflected in the table above:

- **Preserve representation**: Map the DDS unsigned type to a Java signed type of the same size. Designers can be confident that every value in the range of the DDS type has an equivalent value in the range of the Java type. However, logical values will not be preserved in all cases: for example, large unsigned (positive) values will appear as negative values to Java applications.
- **Preserve logical value**: Map the DDS unsigned type to the next-larger Java signed type such that all values in the range of the DDS type can be reflected faithfully in the range of the Java type. However, applications must be prepared to deal with failures that may occur when data values that are logically unsigned mistakenly take a negative value that cannot be faithfully represented on the DDS network.

9.4 Collections

[DDS-XTypes] recognizes three categories of collections: strings (variable-length lists of narrow or wide characters), sequences (variable-length lists of any single element type), and maps (homogeneously typed key-value mappings).

9.4.1 Strings

DDS strings, whether of narrow or wide characters, are represented by Java `String` objects.

- If a string is to be of narrow characters (the default), each Java character shall be truncated to its least-significant byte.
- If a string is to be of wide characters (in which case it must be so marked with `@SerializeAs`), each Java code point shall become a single DDS wide character.

9.4.2 Maps

Any object whose class implements the interface `java.util.Map` shall be considered a DDS map unless marked otherwise with `@SerializeAs`.

9.4.3 Sequences and Arrays

Any object whose class implements the interface `java.util.Collection` shall be considered DDS sequences unless marked otherwise with `@SerializeAs`. If the class implements `java.util.List`, the order of the elements in the sequence shall corresponds

exactly to the order of the elements in the list. Otherwise, the order of the elements in the sequence shall correspond to that returned by the collection's iterator.

Objects of array types shall be considered DDS sequences unless marked otherwise with `@SerializeAs`.

Any Java collection or array may be designated as a DDS array with `@SerializeAs`.

Design Rationale (non-normative)

Objects of array types must receive special care, because a Java array—like any Java object—is stored by reference only. Therefore, although a given array object itself is not of variable length, the reference to it may be reassigned to point to an array of a different length. Even if the reference does not change, the length of the array pointed to cannot in general be discovered by analysis of the type itself and may vary from object to object of the same type.

9.5 Aggregated Types

[DDS-XTypes] recognizes two kinds of aggregated types: structures and unions.

Any DDS type that is not a nested type (in the sense of that word defined by [DDS-XTypes], as indicated in this Type Representation by the annotation `@Nested`) must define a no-argument constructor for use by the Service implementation. Service implementations shall have the capability to invoke this constructor reflectively, even if it is not public.

The fields in the DDS structured type shall correspond to those of the Java class. Their order shall be that returned by the method `java.lang.reflect.Class.getDeclaredFields`. `Static` and/or `transient` fields shall be omitted. Service implementations shall have the capability to get and set the values of fields reflectively regardless of their declared access level (e.g. `public`, `protected`, `private`).

Service implementations need not address the following cases:

- A Java Security Manager (`java.lang.SecurityManager`) prevents privileged access to a non-`public` field or constructor.
- A field that is neither `static` nor `transient` is declared `final`, preventing its value from being modified.
- Object references form a cycle. (Cycles are not permitted by the DDS Type System.)

9.5.1 Structures

Every Java class that is not a collection or map shall be considered a structure by default.

9.5.1.1 Inheritance

Java class extension shall map to structure inheritance in the DDS Type System [DDS-XTypes], subject to the restrictions documented by the `java.io.Serializable` interface, such as those pertaining to non-`Serializable` base types.

9.5.1.2 Extensibility

The extensibility kind shall be determined in the following manner:

- `FINAL`: If the class extends `java.lang.Object` directly and is final, or if explicitly indicated.
- `EXTENSIBLE`: In all other cases, by default, or if explicitly indicated.
- `MUTABLE`: Only if explicitly indicated.

9.5.2 Unions

Any class may be annotated as a union with `@SerializeAs`.

- Such a class must annotate exactly one field to be the discriminator with `@UnionDiscriminator`.
- All other fields that are not `transient` or `static` must be annotated with `@UnionMember`, which shall identify the discriminator value associated with that field.

9.6 Enumerations and Bit Sets

By default, any Java enumeration class will be considered to be a DDS enumeration.

As in IDL, a type that is syntactically an enumeration may be annotated as a bit set type. In this case, objects of these types must also be annotated in order to be serialized correctly. A type member of type `java.util.EnumSet` or `java.util.BitSet` will be serialized as a bit set if marked with `@BitSet`.

9.7 Modules

Each segment of a Java type's package name shall correspond to a module in the DDS Type System [DDS-XTypes]. For example, a class `com.acme.project.TheClass` would be in the nested modules `com::acme::project`.

9.8 Annotations

This Type Representation ignores Java annotation types by default. Java annotations that are intended to be represented explicitly within the DDS Type System must be so annotated with `@SerializeAs`.

Annex A: Java JAR Library File

In addition to this document, this specification includes a Java Archive (JAR) library, `omgdds.jar`. This library contains compiled Java `*.class` files for all of the classes and interfaces specified by this PSM.

This library comprises the compile-time portion of this specification: users shall be able to compile their PSM-compliant code against this library and then deploy the result against any conformant implementation.

Distributors of binary implementations of this PSM may elect to distribute the `omgdds` library alongside their implementation libraries or to package both the contents of `omgdds.jar` and their implementation into a single library.

Annex B: Java Source Code

In addition to this document, this specification includes the Java source code to all of the classes and interfaces specified by this PSM in the zip archive `omgdds_src.zip`. This source code, in the directory `srcJava` within the archive, corresponds to the binary distribution found in the library `omgdds.jar` and is also normative with respect to both its programming interfaces and its embedded documentation comments.

For the convenience of both implementers and application developers, the archive contains additional files that are neither API source code nor documentation. These file are non-normative and include:

- **Code examples**: Short code segments, intended to be illustrative to application developers, can be found in the directory `srcJavaExample` within the archive.
- **build.xml**: A build script, compatible with version 1.6 of the Apache Ant tool[5], can be found in the top-level directory of the archive. It is capable of creating both the `omgdds.jar` and `omg_src.zip` files
- **Project files**: Project definition files compatible with version 3.5 of the Eclipse IDE for Java can be found in the top-level directory of the archive.

[5] See http://ant.apache.org.