UML™ Profile for CORBA™ Specification

Version 1.0
April 2002

ISSUE REPORTING

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 & Specifications, Report a Bug/Issue.

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows:

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction*.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility.*

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.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.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

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

Acknowledgments

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

- Data Access Corporation
- DSTC
- Genesis Development Corporation
- Hewlett Packard
- Inline Software Corporation

- International Business Machines Inc.
- Lucent Technologies, Inc.
- Open IT
- Persistence Software
- Sintef
- Telelogic AB
- UBS AG
- Unisys Corporation

Overview 1

Contents

This chapter contains the following topics.

This specification is based on Rational's "Rose CORBA," which is part of Rational Rose 98i Enterprise. The specification enhances the Rose CORBA specification by aligning it with the UML metamodel and with the working definition of a UML Profile provided by the OMG Business Object Initiative.

1.1 Goal

The UML Profile for CORBA specification was designed to provide a standard means for expressing the semantics of CORBA IDL using UML notation and thus to support expressing these semantics with UML tools.

When one wishes to represent a CORBA type via UML notation, the usual approach is to model it as a Classifier and to stereotype the Classifier to indicate whether it represents an interface, or a valuetype, or a struct, or a union, etc. This is a legitimate approach, since a Stereotype is one of UML's official extension mechanisms. Up to now, however, there has been no *standard* set of extensions of UML for this purpose.

1.2 Scope

The UML Profile for CORBA described in this specification permits the expression of OMG IDL:

```
interface A {};
interface B
{
    attribute A myA;
};
```

It is not possible to express via OMG IDL whether the actual value of **myA** may ever be empty.

Now consider the UML class diagrams Figure 1-1 and Figure 1-2, both of which use the stereotype **<<CORBAInterface>>** defined later in this document. Both of these class diagrams map to the OMG IDL shown above, yet they do not have the same semantics, since one says that an empty value for **myA** is permitted and the other says it is not.

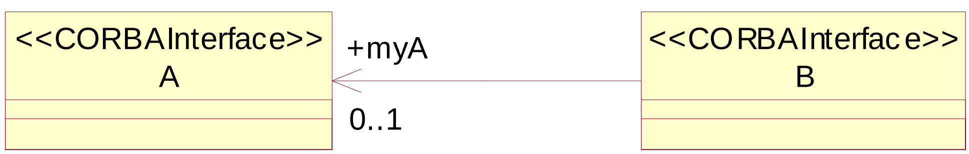

Figure 1-1 Empty Value Permitted

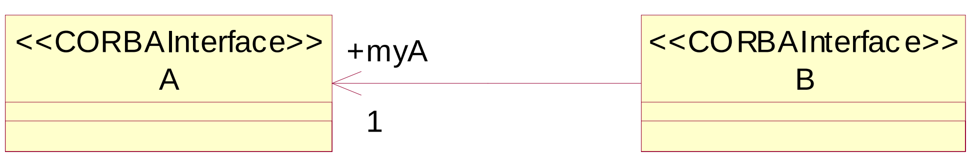

Figure 1-2 Empty Value Not Permitted

Unless ruled out by explicit constraints, all properties of UML metamodel elements contained in the UML Profile may be used to express an object model that conforms to the profile. For example, it is permissible to use UML facilities for expressing aggregation properties of Associations. The modeler may specify invariants for Classes and pre and postconditions of Operations.

The submitters' vision is that such permitted semantic specifications that go beyond what is expressible in OMG IDL will be considered normative aspects of future official OMG specifications.

1.3 Specific Design Decisions

1.3.1 Namespace Containment

In CORBA declarations are always contained in some namespace scope with nesting that allows some declarations to be contained by a container that is in turn contained, and so on. An anonymous global scope is available for declarations that are not inside a module. The two main Namescopes are modules that act purely as a container and may be nested arbitrarily deep, and interfaces that act as a unit of functionality represented at runtime by a CORBA Object as well as a container for data type declarations. Interfaces may not contain other interfaces. Constructed data types (structs and unions) also act as namespaces for their member elements, which may in turn be in-line declarations of nested (contained) constructed data types.

In UML, Namespace is an abstract meta-Class inherited by many other meta-Classes that need to contain other named ModelElements. The Namespaces that concern us for modeling CORBA are Package (from Model Management) and Classifier (from Core). The notation used to depict Namespace containment in UML is the "circle-plus," (UML 1.3, Section 3.13.2) which is used to represent all CORBA Namespace containment, except for module containment of data type and interface declarations, which is represented using the usual Package box surrounding the ModelElements that it contains.

Unfortunately some tools don't support the circleplus notation, but most will have some mechanism for representing Namespace containment, and this should be used until a conformant version can be produced.

1.3.2 Using Associations to Represent User-Defined Types

The aggregation of members into constructed types in CORBA is always modeled as an aggregation Association with navigability away from the aggregate.

The name of the part is always modeled as the role name of the part in the Association.

All CORBA data types (here we mean non-object types, where object types are interfaces and value types supporting interfaces) must be fully instantiated in order to be passed as parameters or return values. The only nulls in CORBA are nil object references and value types. Therefore the multiplicities for part AssociationEnds in aggregation Associations must be 1..1 for all non-object types. Modelers may specify multiplicities of 0..1 when nil objects are valid, or 1..1 when nil objects are not valid.

The multiplicities for the aggregate AssociationEnd will usually be 0..1, indicating that the part type will be owned by at most one aggregate, but that it may be instantiated independently of an aggregate.

In the case where the part is a CORBA interface, the multiplicity may be 0..*, as the object reference may be a part of many user-defined types. (This is the default for mapping from OMG IDL - but modeling in UML allows the modeler to constrain the multiplicity further.) For object types the default is weak aggregation.

In the case where a new part type is declared within the scope of another user-defined type, CORBA semantics dictate that the part type cannot be instantiated independently of the aggregate, and therefore the multiplicity at the aggregate AssociationEnd must be 1..1.

Profiles and Virtual Meta Models 2

Contents

This chapter contains the following topics.

2.1 General Definition of a UML Profile

Currently, there is no normative definition of a UML profile. However, the Business Object Initiative RFPs elucidated the following working definition of a UML profile.

A *UML profile* is a specification that does one or more of the following:

- Identifies a subset of the UML metamodel (which may be the entire UML metamodel).
- Specifies "well-formedness rules" beyond those specified by the identified subset of the UML metamodel. "Well-formedness rule" is a term used in the normative UML metamodel specification (ad/99-06-08) to describe a set of constraints written in natural language and UML's Object Constraint Language (OCL) that contributes to the definition of a metamodel element.
- Specifies "standard elements" beyond those specified by the identified subset of the UML metamodel. "Standard element" is a term used in the UML metamodel specification to describe a standard instance of a UML stereotype, tagged value, or constraint.
- Specifies semantics, expressed in natural language, beyond those specified by the identified subset of the UML metamodel.

- Specifies common model elements; that is, instances of UML constructs expressed in terms of the profile.

2.2 Virtual Metamodel of Stereotypes

The UML specification makes the following comment in its discussion of Stereotypes:

> *The stereotype concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs.*[1]

This section presents a virtual metamodel for the Stereotypes defined by the UML Profile for CORBA. The UML specification provides only a few guidelines for how to express a virtual metamodel, and the UML Profile for CORBA is the first profile for which an RFP was issued by the OMG. Thus, the submitters have worked out an approach to describing the virtual metamodel that hopefully will establish some precedents that will be able to be leveraged by the architects of future UML profiles.

2.2.1 Background Facts

2.2.1.1 Legal Relationships Among Stereotypes

In the UML metamodel, a Stereotype is a GeneralizableElement. Thus it is legal to define Generalization (inheritance) Relationships among Stereotypes. Furthermore, a GeneralizableElement is a ModelElement, and Dependency Relationships can be defined among ModelElements. Thus it is legal for Stereotypes to participate in Dependency Relationships. However, a Stereotype is not a Classifier. Therefore, Stereotypes may not participate in Association Relationships.

2.2.1.2 The Notion of Extension

In the UML metamodel, a Stereotype *extends* an element or elements of the metamodel. For example, the Stereotype <<CORBAException>> extends the UML metamodel Exception element.

2.2.1.3 Constraints

All of the Constraints defined for a Stereotype in this specification are intended to describe Constraints on the stereotyped ModelElements.

1.[UML 1999] section 2.6.

2.2.1.4 Abstract Stereotypes

Some abstract Stereotypes are defined and, in keeping with UML notation, abstractness is denoted by italicizing the Stereotype's name. In UML an abstract GeneralizableElement cannot be instantiated. The abstract Stereotypes are useful for avoiding repetition in multiple Stereotypes that logically have common properties.

2.2.1.5 Common Model Elements

The common model elements contained in the profile are all instances of UML DataType and Class that are stereotyped as <<CORBAPrimitive>>. These include an instance of DataType for each of the CORBA basic types, and an instance of Class for each of CORBA::Object and CORBA::baseValue. We place these model elements in a package called *CORBA*.

2.2.2 Using UML Notation for Virtual Metamodeling

In light of these facts, the specification takes the following approach to using UML notation to express the virtual metamodel:

- The model is expressed via class diagrams.
- Each Stereotype plays the client role in a Dependency Relationship with the UML metaclass that it extends. These Dependencies are stereotyped <<baseElement>>. We use this as non-standard notation because relationships afford greater clarity than TaggedValues.
- Each Stereotype is expressed via a Classifier box, even though a Stereotype is not a Classifier. The keyword "<<stereotype>>" does NOT represent a stereotype itself--it is simply a notational marker for the underlying Stereotype metaclass.
- Generalization Relationships among Stereotypes are expressed in the standard UML fashion.

2.2.3 Constraints

Constraints are expressed in English and OCL.

2.2.3.1 OCL Convenience Operations Reused from UML 1.3

The OCL for the formal constraints reuses the following OCL convenience operations defined for the Classifier metaclass by UML 1.3 [UML 1999, Section 2.5.3].

- Operation [4] **allAttributes**
- Operation [5] **associations**
- Operation [7] **oppositeAssociationEnds**

2.2.3.2 Additional OCL Convenience Operations for UML Metamodel Elements

The OCL convenience operations in this section can be applied generally to UML 1.3 and are not specific to the UML Profile for CORBA. However, they are defined in order to produce more compact and readable OCL.

For ModelElement

[1] The operation **allStereotypes** results in a Set containing the ModelElement's Stereotype and all Stereotypes inherited by that Stereotype (as opposed to all Stereotypes inherited by the ModelElement).

```
allStereotypes : Set(Stereotype);
allStereotypes = self.stereotype->union
   (self.stereotype.generalization.parent.allStereotypes)
```

[2] The operation **isStereotyped** determines whether the ModelElement has a Stereotype whose name is equal to the input name.

```
isStereotyped : (stereotypeName : String) : Boolean;
self.stereotype.name = stereotypeName
```

[3] The operation **isStereokinded** determines whether the ModelElement has a Stereotype whose name is equal to the input name or if it has a Stereotype one of whose ancestors' name is equal to the input name.

```
isStereokinded : (stereotypeName : String) : Boolean;
self.allStereotypes->exists (stereotype | stereotype.name = stereotypeName)
```

There are some OCL convenience operations defined in this specification that apply more narrowly to certain extensions of UML that the profile defines. These operations appear inline with the Constraints for those specific extensions.

For Classifier

[1] The operation **navigableOppositeEnds** results in a Set containing all navigable AssociationEnds that are opposite to the Classifier.

```
navigableOppositeEnds : Set(AssociationEnd);
navigableOppositeEnds = self.oppositeAssociationEnds
                          ->select(end | end.isNavigable)
```

[2] The operation **allEnds** results in a Set containing all AssociationEnds for which the Classifier is the type.

```
allEnds : Set(AssociationEnd);
allEnds = self.associations->collect(assoc | assoc.connection)
```

[3] The operation **nonNavigableNearEnds** results in a Set containing all AssociationEnds that are adjacent to the Classifier and that are non-navigable.

```
nonNavigableNearEnds : Set(AssociationEnd);
nonNavigableNearEnds = self.allEnds->select(end | end.type = self
                                     and not end.isNavigable)
```

[4] The operation **navigableEnds** results in a Set containing all navigable AssociationEnds for which the Classifier; that is, self is the type.

```
navigableEnds : Set(AssociationEnd);
navigableEnds = allEnds->select(end | end.isNavigable)
```

CORBA Profile Definition 3

Contents

This chapter contains the following topics.

3.1 Introduction

This chapter is the normative definition of the CORBA Profile of UML. It consists of a virtual metamodel, showing extensions to UML using the notation described in the previous chapter. This is followed by a description of the way in which each IDL construct is modeled, including TaggedValues, Stereotypes, and Constraints used to model it, and any notation that differs from the standard UML representation of the Stereotype's baseElement.

3.2 Structure of the Profile

As described in Section 2.1, "General Definition of a UML Profile," on page 2-1, a Profile consists of the following:

- An identified subset of the UML Meta-model. This is addressed in Section 3.3, "Identified Subset of UML," on page 3-2.

- Specifications of Standard Elements (Stereotypes, TaggedValues, and Constraints). The stereotypes are shown in a virtual metamodel in Section 3.4, "The Virtual Metamodel," on page3-3, and then explained in detail in Sect i on3.5, "The CORBA Type Representations," on page 3-10. All the Standard Elements as well as additional well-formedness rules are specified in Section 3.5, "The CORBA Type Representations," on page 3-10.
- Specifications of semantics in natural language. These are given in Section 3.5, "The CORBA Type Representations," on page 3-10.
- Specifications of Common ModelElements in terms of the Profile. This Profile defines a number of CORBA-specific type primitives in the package "CORBA." These are defined in Section 3.5.1, "CORBA Basic Types," on page 3-11.

The Standard Elements are defined within a package called "CORBAProfile."

3.3 Identified Subset of UML

The CORBA Profile extends the following standard UML packages:

- Core
- Common Behavior
- Model Management

The following concrete metaclasses, and implicitly all super-metaclasses of these metaclasses, are used:

From Core:

- Abstraction
- Association
- AssociationEnd
- Attribute
- Binding
- Class
- Comment
- Constraint
- DataType
- Dependency
- ElementOwnership
- Generalization
- Operation
- Parameter
- Permission

- Usage

From Common Behavior:

- Exception

From Model Management:

- ElementImport
- Package

3.4 The Virtual Metamodel

Figure 3-2 on page 3-4 to Figure 3-11 on page 3-10 describe the hierarchy of Stereotypes that model CORBA IDL. The semantics of these Stereotypes is given in Section 3.5, “The CORBA Type Representations,” on page 3-10.

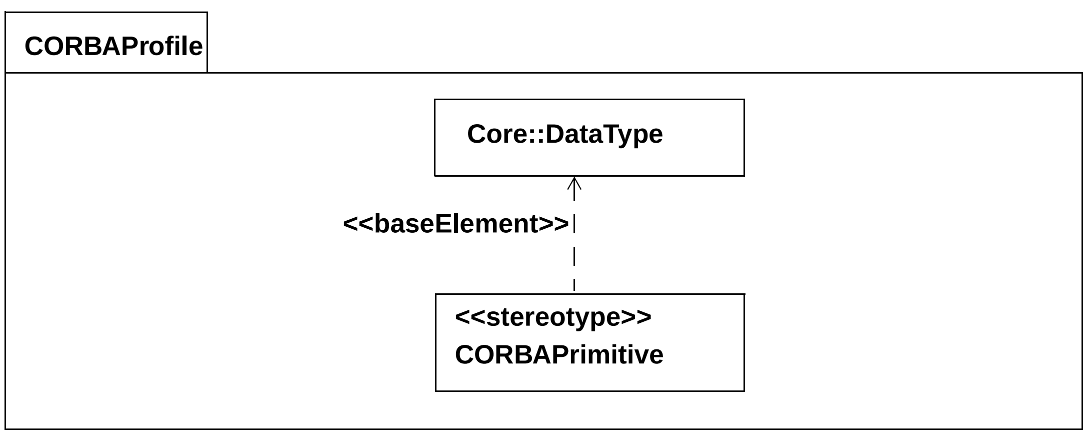

Figure 3-1 Virtual Metamodel for CORBA Primitives

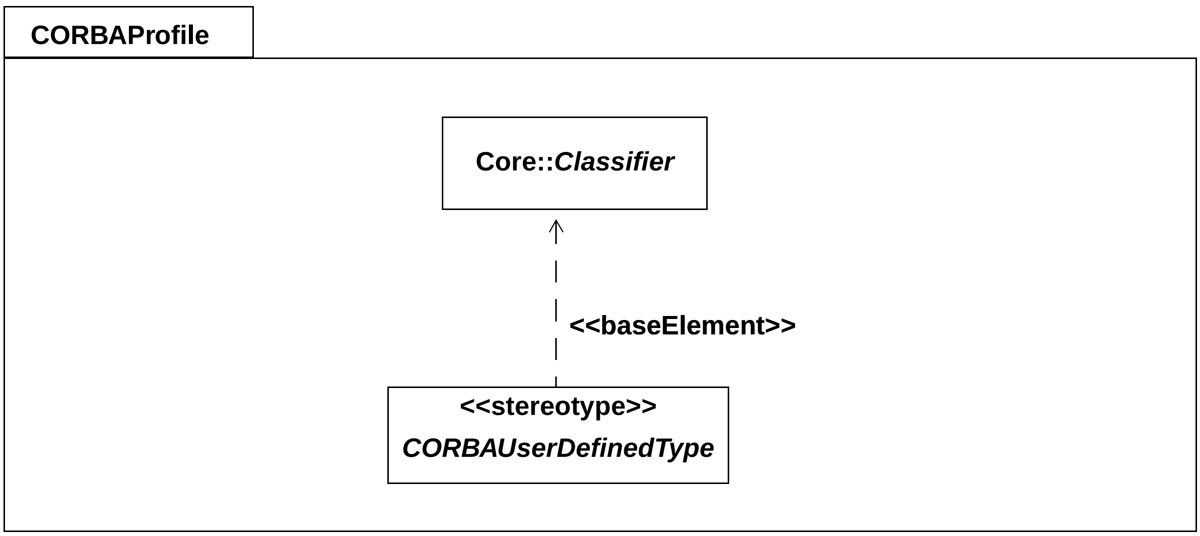

Figure 3-2 Virtual Metamodel for CORBA User-Defined Types

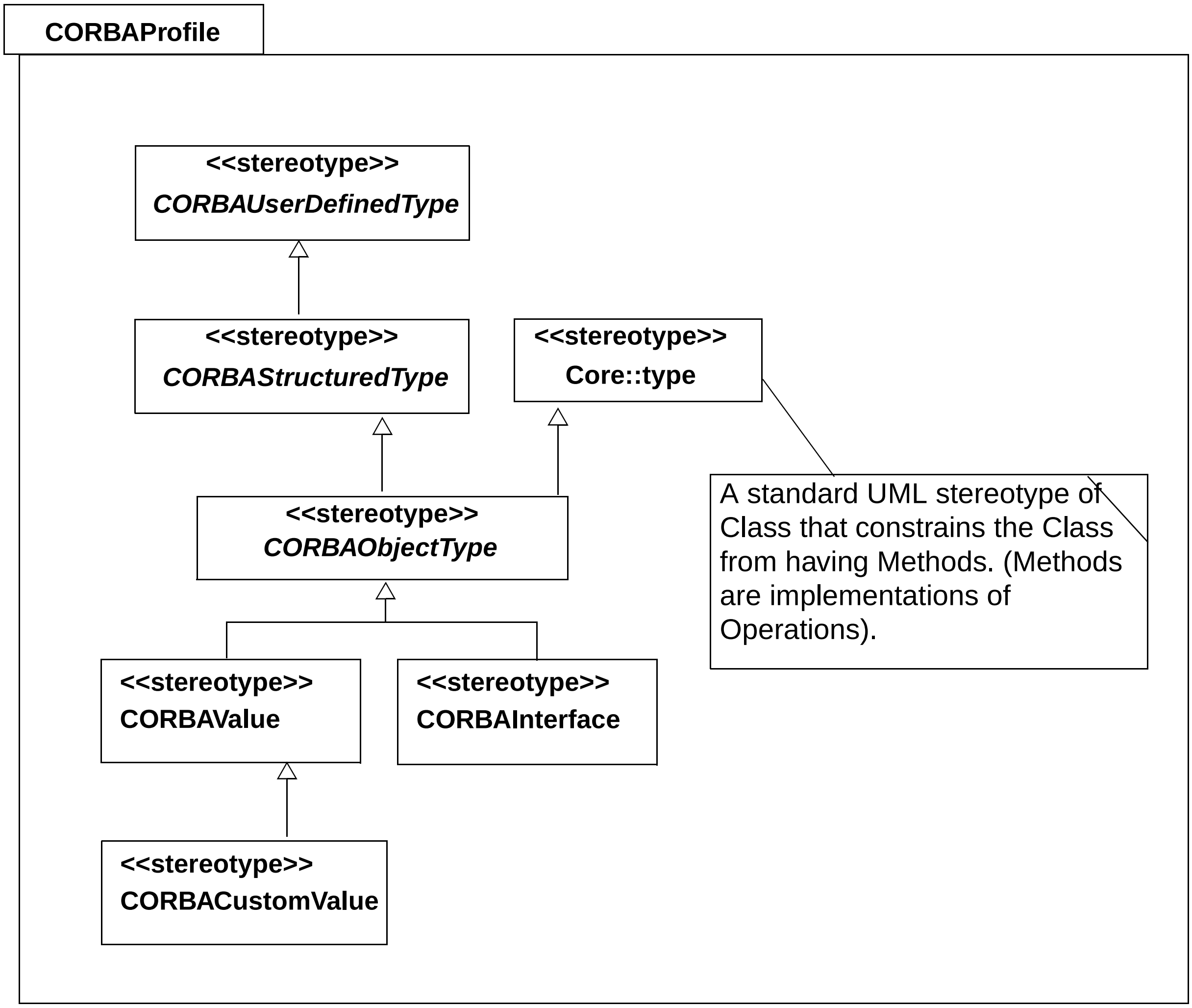

Figure 3-3 Virtual Metamodel for CORBA Object Types

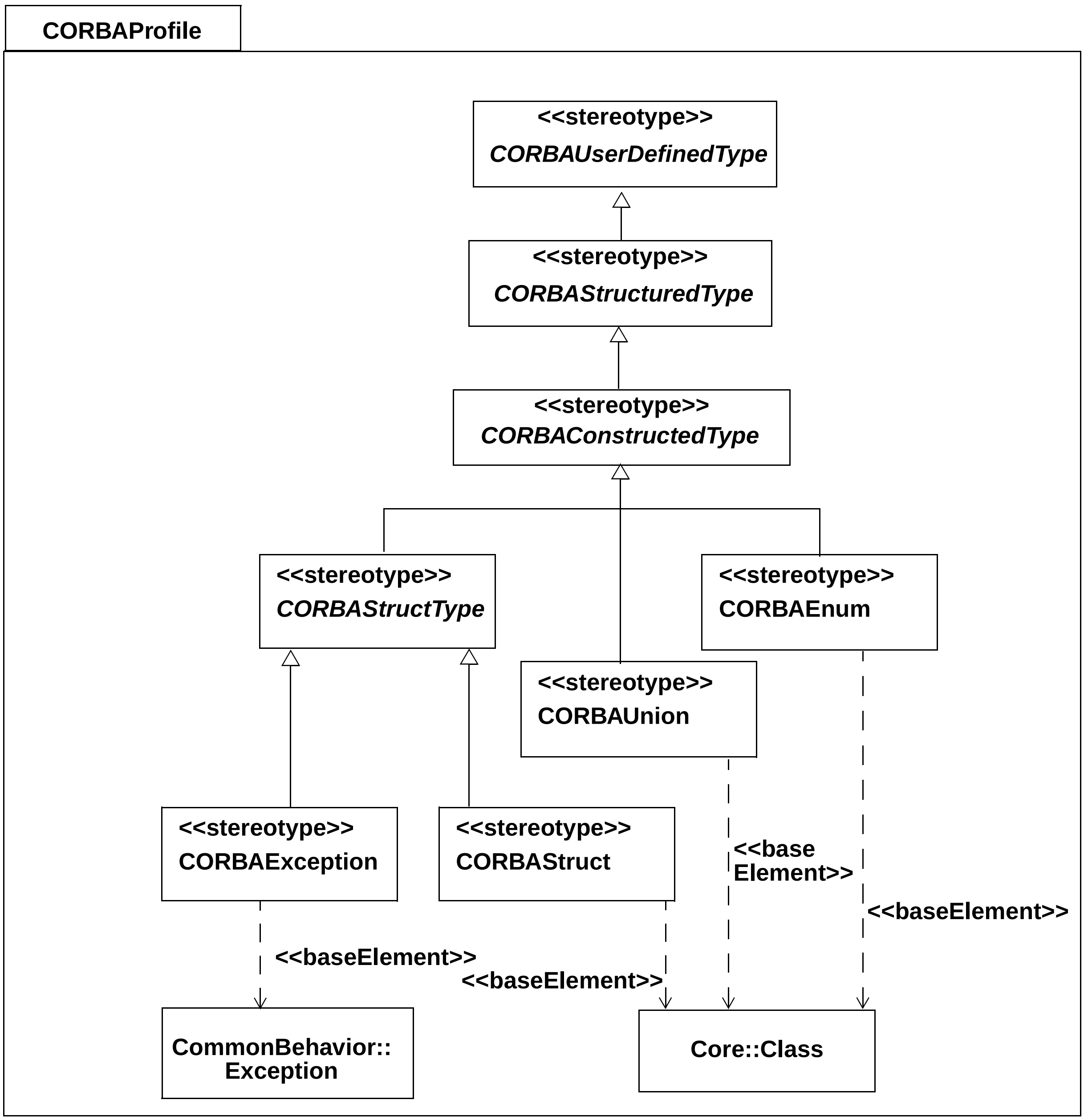

Figure 3-4 Virtual Metamodel for CORBAConstructed Types

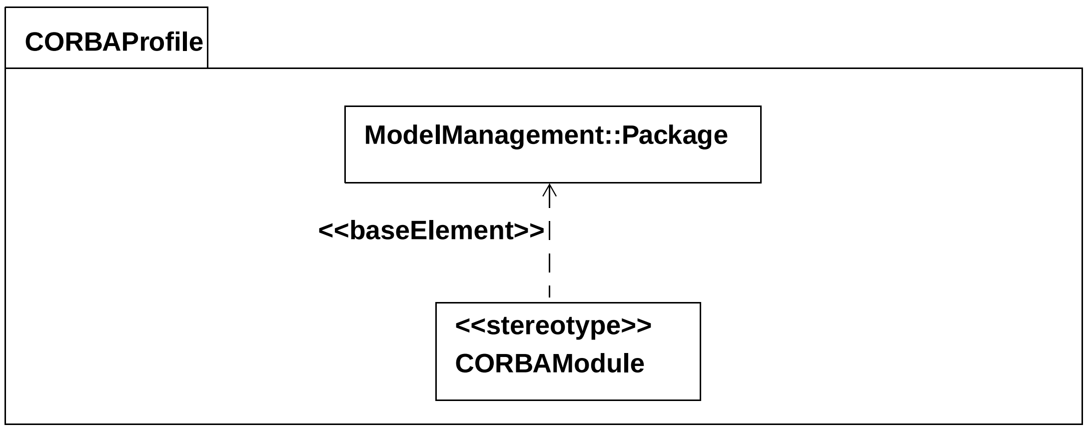

Figure 3-5 Virtual Metamodel for CORBA Module

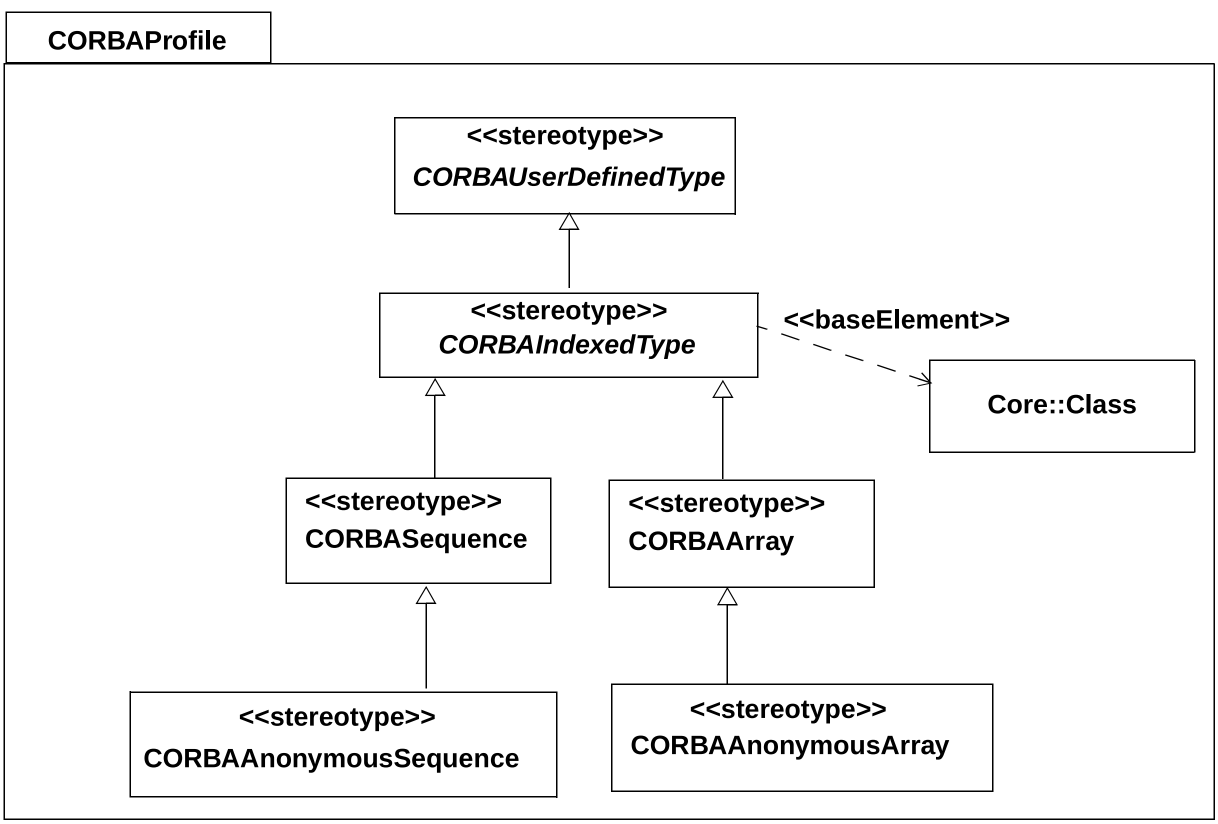

Figure 3-6 Virtual Metamodel for CORBA Indexed Types

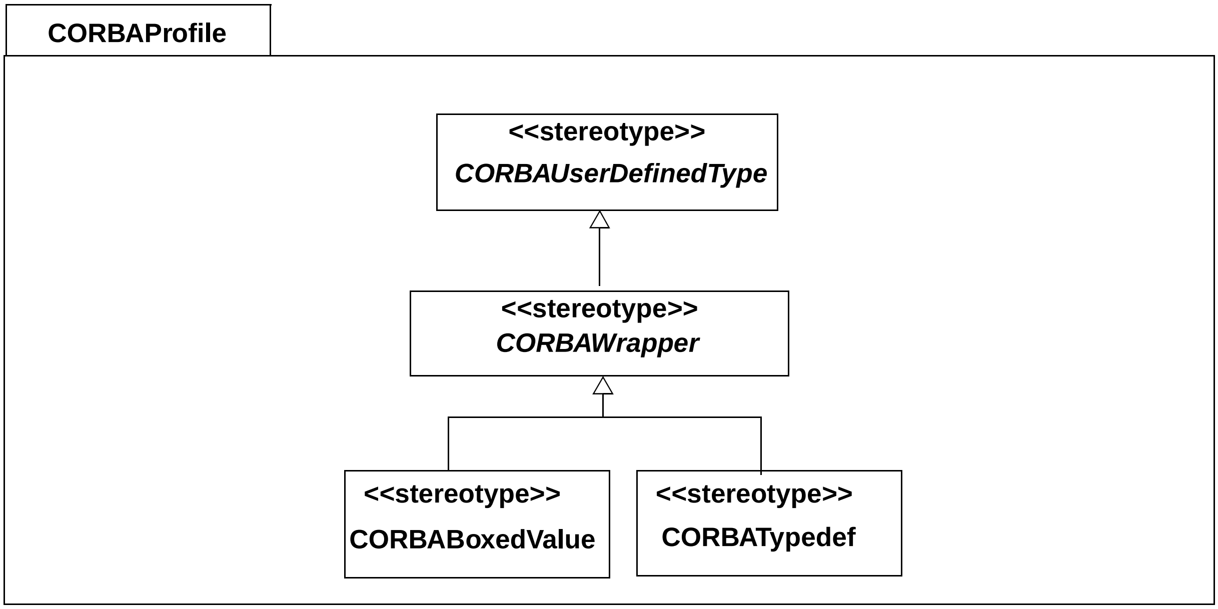

Figure 3-7 Virtual Metamodel for CORBA Wrapper Types

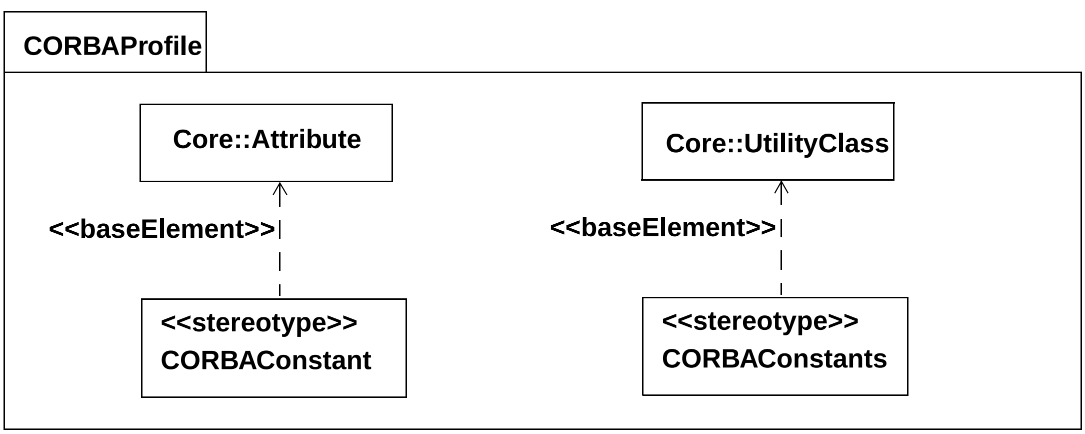

Figure 3-8 Virtual Metamodel for CORBA Constants and Their Container

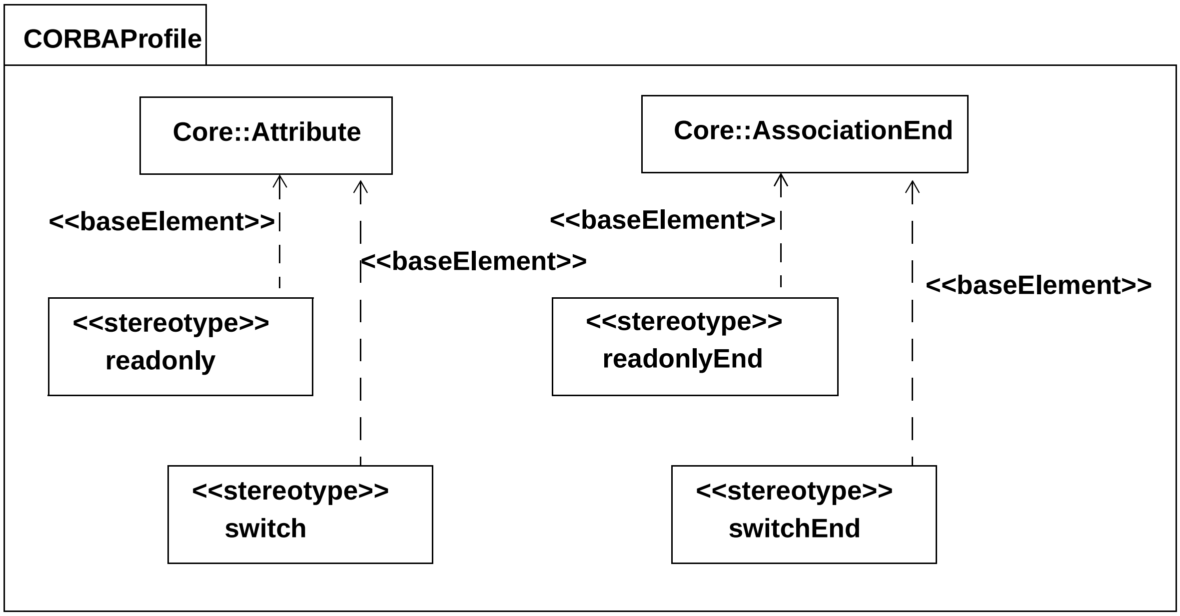

Figure 3-9 Virtual Metamodel for Stereotypes of Attribute and AssociationEnd

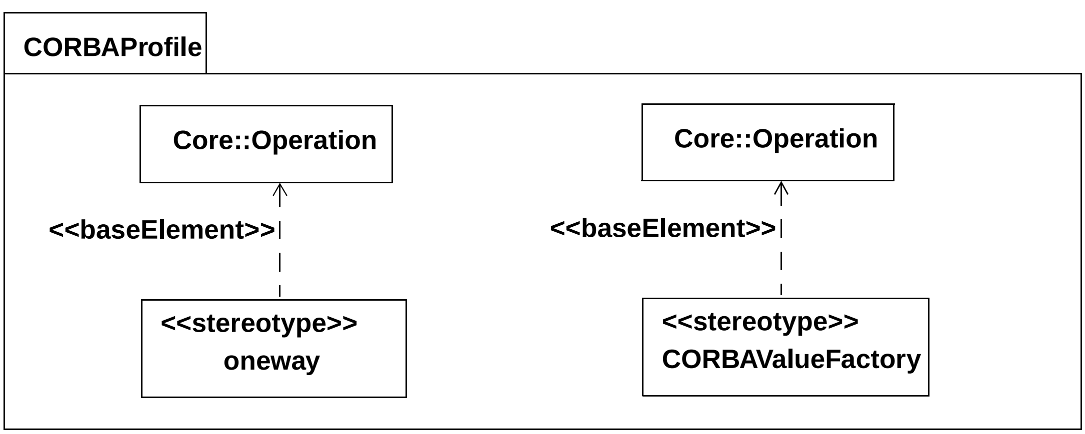

Figure 3-10 Virtual Metamodel for Stereotypes of Operation

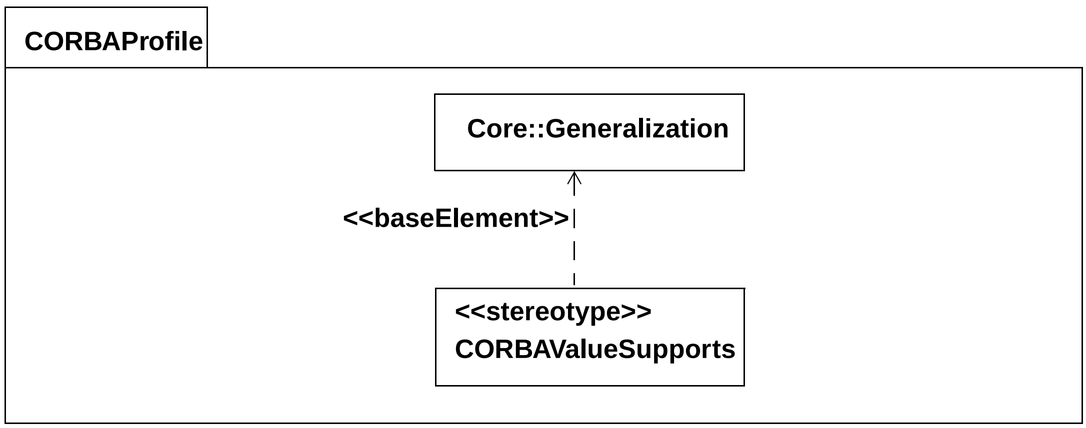

Figure 3-11 Virtual Metamodel for CORBA Values Supporting Interfaces

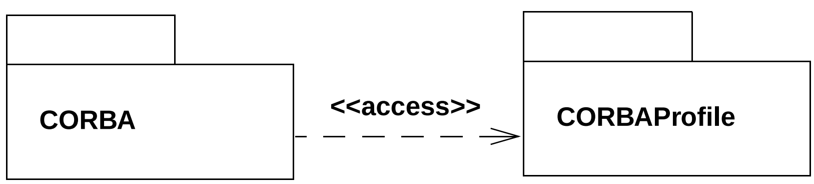

Figure 3-12 Dependencies between CORBA and CORBAProfile

3.5 *The CORBA Type Representations*

This section describes all the Stereotypes introduced in the Virtual Metamodel, and adds the necessary TaggedValues, Constraints, and Common Model Elements to complete the Profile. The subsections are arranged to match the structure of the Virtual Meta Model.

3.5.1 CORBA Basic Types

3.5.1.1 UML Standard Elements

Stereotypes and Tagged Values

The CORBA basic types are represented by UML DataTypes with the <<CORBAprimitive>> stereotype in the "CORBA" package. This package also contains the base types for CORBA interfaces and value types.

Constraints

Common ModelElements

The following <<CORBAprimitive>>-stereotyped UML DataTypes are introduced in the package "CORBA." Their semantics is defined in Chapter 3, "OMG IDL Syntax and Semantics," of the *CORBA/IIOP Specification*.

- short
- long
- long long
- double
- long double
- unsigned short
- unsigned long
- unsigned long long
- any
- boolean
- string
- octet
- void
- char
- wchar
- float
- wstring
- typecode
- native

The CORBA package also contains a Class "Object," stereotyped as <<CORBAInterface>>, and a Class "ValueBase," stereotyped as <<CORBAValue>>.

3.5.2 CORBA User-defined Types

3.5.2.1 UML Standard Elements

Stereotypes and Tagged Values

The abstract stereotype <<*CORBAUserDefinedType*>> is the base for all the concrete stereotypes representing IDL declarations.

The ability to choose a RepositoryId for any scoped name in IDL using typeId declarations is modeled as a TaggedValue { typeId = *repository-id* }, which may be attached to any UML ModelElement representing a CORBA type declaration.

The ability to choose a RepositoryId prefix for declarations inside any scoped name representing an IDL namespace using typePrefix declarations is modeled as a TaggedValue { typePrefix = *prefix* }, which may be attached to any UML ModelElement representing a CORBA namespace.

Constraints

***CORBAUserDefinedType* (Core::*Classifier*) and CORBAModule (ModelManagment::Package)**

[1] All model elements representing CORBA Namespaces may not directly contain another element of the same name.

```
self.ownedElements
->forAll(ownedEl | not ownedEl.name = self.name)
```

CORBAUserDefinedType (Core::Classifier)

[1] All Attributes of a <<CORBAUserDefinedType>>-stereotyped Classifier that are not stereotyped <<CORBAConstant>> must be of a type that is stereotyped <<CORBAPrimitive>>, and for which the ownerScope is “instance,” the targetScope is “instance,” and the changeability is “changeable.”

Note – “Changeable” here is different than IDL readonly, since the value of even a readonly attribute can change (it just can't be changed via the CORBA interface).

```
self.allAttributes
->forAll(attribute | not attribute.isStereotyped("CORBAConstant") implies
    attribute.type.isStereotyped("CORBAPrimitive") and
    attribute.ownerScope = #instance and
    attribute.targetScope = #instance and
    attribute.changeability = #changeable
```

[2] All Associations in which a <<CORBAUserDefinedType>>-stereotyped Classifier participates that have navigable opposite AssociationEnds must be binary and unidirectional.

```
self.navigableOppositeEnds
->forAll(end | end.association.connection->size = 2 and
                     end.association.connection
                     ->select(end | end.isNavigable)->size = 1)
```

[3] All navigable opposite AssociationEnds of a <<CORBAUserDefinedType>>-stereotyped Classifier must have changeability "changeable," aggregation "none," targetScope "instance," and a type that is stereotyped with a descendant of <<CORBAUserDefinedType>> or stereotyped <<CORBAPrimitive>>.

```
self.navigableOppositeEnds
->forAll (end | end.changeability = #changeable and
             end.aggregation = #none and
             end.targetScope = #instance and
             (end.type.isStereokinded("CORBAUserDefinedType") or
                 end.type.isStereotyped("CORBAPrimitive")))
```

[4] All non-navigable near AssociationEnds of a <<CORBAUserDefinedType>>-stereotyped Classifier must have targetScope "instance."

```
self.nonNavigableNearEnds.targetScope = #instance
```

[5] All Associations in which a <<CORBAUserDefinedType>>-stereotyped Classifier participates that have a navigable opposite AssociationEnd whose type is not a <<CORBAInterface>>-stereotyped Classifier must have a near AssociationEnd with the aggregation "composite."

Note – Composite aggregation implies a multiplicity upper bound of 1.

```
self.navigableOppositeEnds
->forAll(opEnd | not opEnd.type.isStereotyped("CORBAUserDefinedType")
implies
    opEnd.association.connection
    ->select(end | end <> opEnd).aggregation = #composite)
```

[6] A <<CORBAUserDefinedType>>-stereotyped Classifier cannot participate in any AssociationClasses.

```
self.associations->forAll(assoc | not assoc.oclIsTypeOf(AssociationClass))
```

3.5.2.2 Notation

Association and Aggregation

The notation used for aggregation of elements has a diamond at the aggregate AssociationEnd, and an arrow at the part AssociationEnd. Unless the part is a CORBA interface type with a lifecycle independent of the aggregate, the strong aggregation (composition) black diamond is used. The default Association for interface types does not aggregate, and its multiplicity is 0..* at the near end.

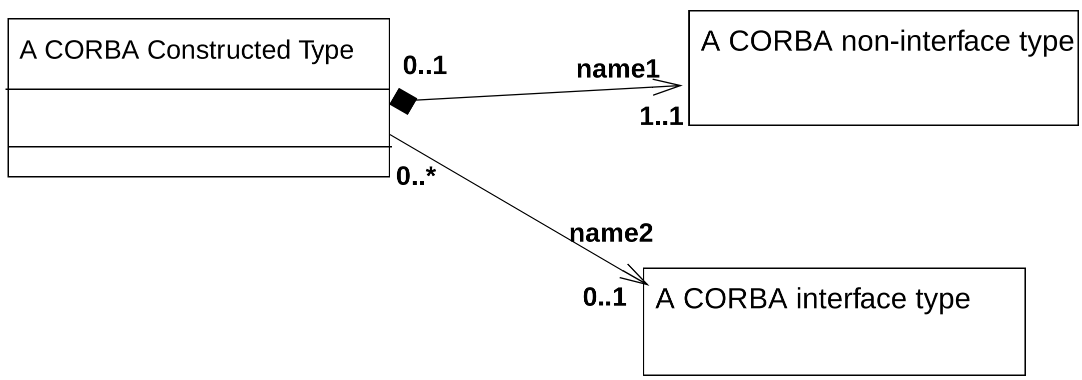

Figure 3-13 Aggregate Notation for CORBA Constructed Types

Namespace Containment

The notation for Namespace containment of IDL constructs modeled as Classifiers in this Profile is the "circle-plus" notation as shown in Figure 3-14.

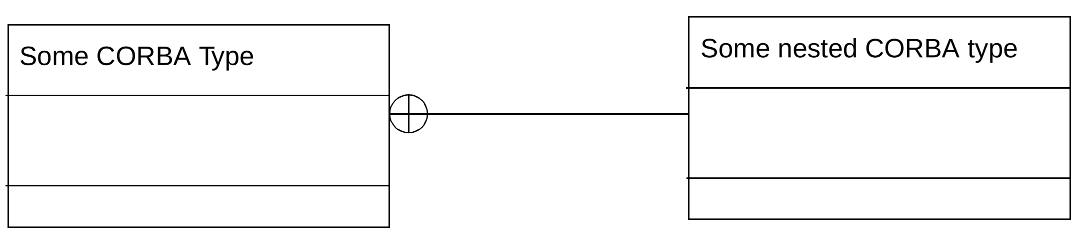

Figure 3-14 UML Namespace Containment Notation for Nested CORBA Constructs

The containment of IDL constructs modeled as Features of Classifiers is indicated in the usual way by graphical containment of the Feature in the Class box, or by aggregation.

3.5.3 CORBA Structured Types

We use the term structured types to refer to CORBA Object types (interfaces and valuetypes) and CORBA constructed types (structs, exceptions, unions, and enums). All of these types define a new name scope containing other declarations. These contained IDL declarations are tagged to retain their order when models are UML derived from IDL, so that equivalent IDL may be generated from the model later.

3.5.3.1 UML Standard Elements

Stereotypes and Tagged Values

The abstract stereotype <<CORBAStructuredType>> specializes <<CORBAUserDefinedType>>, and has two derived abstract stereotypes: <<*CORBAObjectType*>> and <<*CORBAConstructedType*>>.

Constraints

***CORBAStructuredType* : *CORBAUserDefinedType* (Core::Class)**

[1] All Attributes, navigable opposite AssociationEnds, and ownedElements of a <<CORBAStructuredType>>-stereotyped Class must have a tagged value IDLOrder whose values are contiguous integers starting from 0.

```
let featureOrderTags = self.feature
->collect(feature | feature.taggedValue
->select(tag | tag.name = "IDLOrder")) and

let endOrderTags = self.navigableOppositeEnds
->collect(end | end.taggedValue
->select(tag | tag.name = "IDLOrder")) and

let ownedElementOrderTags = self.ownedElements
->collect(ownedElement | ownedElement.taggedValue
->select(tag | tag.name = "IDLOrder")) and

let orderTags
= featureOrderTags->union(endOrderTags->union(ownedElementOrderTags)) and

let orderValues = orderTags->collect(tag | tag.value) and
let numOfOrderValues = orderTags->size in

self.feature->forAll(feature | feature.taggedValue
->select(tag | tag.name = "IDLOrder")->size = 1) and
```

```
self.navigableOppositeEnds->forAll(end | end.taggedValue
->select(tag | tag.name = "IDLOrder")->size = 1) and

self.ownedElements->forAll(ownedElement | ownedElement.taggedValue
->select(tag | tag.name = "IDLOrder")->size = 1) and

orderValues->isUnique(n | n) and
orderValues->forAll(value | value >= 0 and (value <= numOfOrderValues - 1))
```

3.5.4 Module Declaration

3.5.4.1 UML Standard Elements

Stereotypes and Tagged Values

An IDL module is represented by a UML package (from Model Management) stereotyped as <<CORBAModule>>.

IDL module containment (nesting) is modeled by Namespace containment of one <<CORBAModule>>-stereotyped UML package within another.

The ability to choose a RepositoryId for any scoped name in IDL using typeId declarations is modeled as a TaggedValue { typeId = *repository-id* }, which may be attached to any UML package representing a CORBA module.

The ability to choose a RepositoryId prefix for declarations inside any scoped name representing an IDL namespace using typePrefix declarations is modeled as a TaggedValue { typePrefix = *prefix* }, which may be attached to any UML package representing a CORBA module.

Constraints

CORBAModule (ModelManagement::Package)

[1] A <<CORBAModule>>-stereotyped package may directly contain only <<CORBAModule>>-stereotyped packages or Classes stereotyped as <<CORBAConstants>> or as a descendant of <<CORBAUserDefinedType>>.

```
self.ownedElement->forAll(el | el.isStereotyped("CORBAModule") or
                     el.isStereotyped("CORBAConstants") or
                 el.isStereokinded("CORBAUserDefinedType"))
```

[2] A <<CORBAModule>>-stereotyped package may directly contain at most one Class stereotyped as <<CORBAConstants>>.

```
self.ownedElement
->collect(el | el.isStereotyped("CORBAConstants"))->size <= 1
```

3.5.4.2 Notation

The notation for UML package is used, with the additional stereotype label <<CORBAModule>>.

For example the following IDL:

```
module Parent {
    module Child1 {};
    module Child2 {
        module Grandchild {};
    };
};
```

is represented in UML package notation in Figure 3-15.

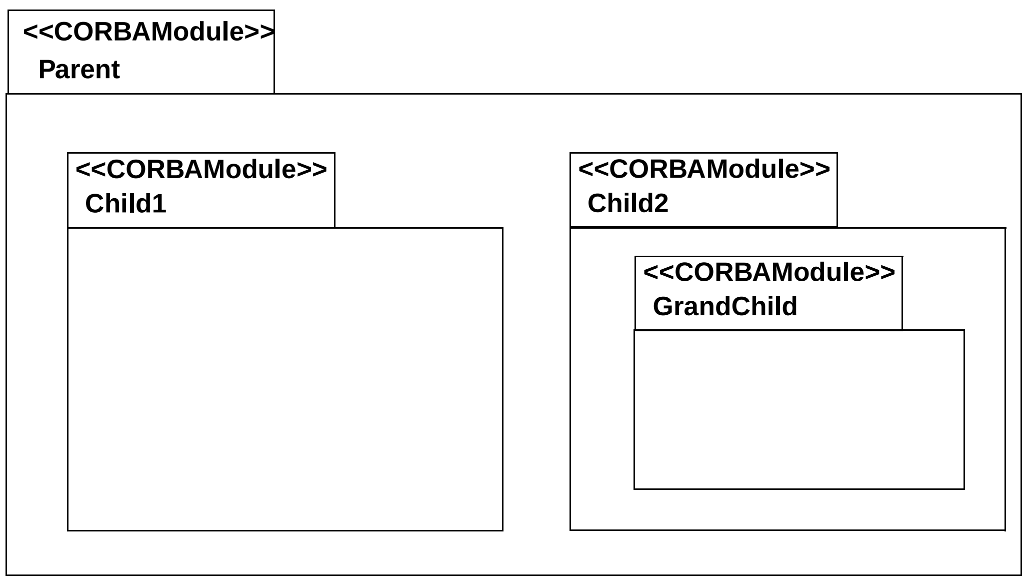

Figure 3-15 Module Package Notation

Modules that contain nested modules may also be represented using Namespace containment notation.

The IDL above is shown using the Namespace containment notation in Figure 3-16 on page 3-18.

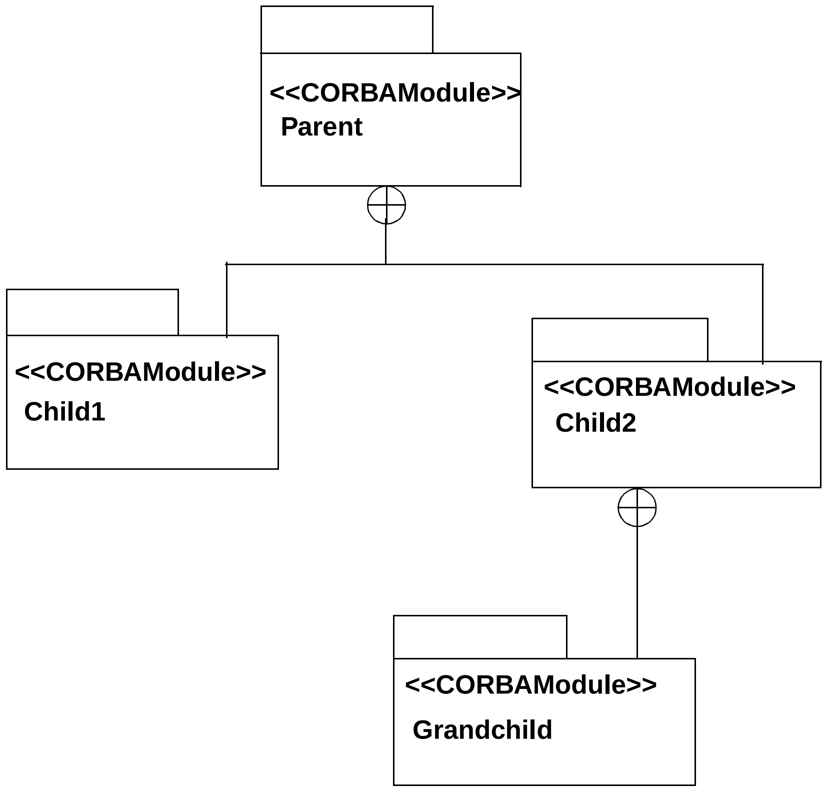

Figure 3-16 Module Namespace Containment Notation

3.5.5 CORBA Object Types

3.5.5.1 UML Standard Elements

Stereotypes and Tagged Values

The abstract stereotype <<*CORBAObjectType*>> is a specialization of UML Class that captures common characteristics of CORBA interfaces and value types.

Constraints

***CORBAObjectType* : *CORBAStructuredType, Core::type* (Core::Class)**

Note – Only object types are descendants of <<type>>, since only object types are explicitly "realized" by implementation classes in CORBA, whereas non-object types are simply mapped into data structures.

---

[1] A <<*CORBAObjectType*>>-stereotyped Class may not have Receptions.

```
self.features->forAll(feature | not feature.oclIsTypeOf(Reception))
```

[2] The ownedElements of a <<*CORBAObjectType*>>-stereotyped Class may only be Classes stereotyped with a descendant of <<*CORBAConstructedType*>>, <<*CORBAIndexedType*>>, or <<*CORBAWrapper*>>.

```
self.ownedElements
->forAll(ownedEl | ownedEl.isStereokinded("CORBAConstructedType") or
                 ownedEl.isStereoKinded("CORBAIndexedType") or
                 ownedEl.isStereoKinded("CORBAWrapper"))
```

[3] All parents of a <<*CORBAObjectType*>>-stereotyped Class must have the same stereotype as the Class.

```
self.generalization
->forAll(generalization | generalization.parent..stereotype.name =
self.stereotype.name)
```

3.5.6 Interface

3.5.6.1 UML Standard Elements

CORBA interfaces are modeled using UML Classes.

Note – The correspondingly named UML metamodel element "Interface" (from Core) is inappropriate for modeling an IDL interface, as it may not have Attributes or Associations that can be navigated from the Interface. Although IDL attributes are implemented as accessor and modifier methods in most language mappings, the IDL attribute is a distinguished type in the CORBA Interface Repository, which is modeled in this Profile by UML Attribute. In addition we require IDL attributes whose type is an aliased or constructed type to be represented by navigable Associations between the UML ModelElement representing the IDL interface and the UML ModelElement representing the IDL attribute's type.

The representation for IDL interface attributes is fully specified in Section 3.5.23, "Attribute," on page 3-57. In summary:

- Attributes whose types are basic types are represented as UML Class Attributes, having syntax specified inline. These Attributes are constrained to have Visibility set to public.
- Attributes whose types are user-defined types are represented as UML Associations between the <<CORBAInterface>>-stereotyped Class and the CORBA Profile ModelElements representing that user-defined type.

The representation for IDL operations is fully specified in Section 3.5.22, "Operation," on page 3-53. In summary:

- Each IDL operation is represented as a UML Class Operation.

The raising of an IDL exception is represented using TaggedValues on Operations and Attributes.

Containment of CORBA data type declarations by the interface's name scope is represented using UML Namespace containment.

Stereotypes and Tagged Values

An IDL interface is represented by a UML Class that is stereotyped <<CORBAInterface>>.

Local interfaces are represented using the TaggedValue { isLocal = TRUE }.

When mapping each semi-colon-separated declaration in an IDL interface to a ModelElement in a UML model the ModelElement will be tagged with the TaggedValue {IDLOrder = *N*}, where *N* is the number of the declaration from zero upwards. Models created directly in UML will also have an IDLOrder tag attached to each declaration ModelElement belonging to a <<CORBAInterface>>-stereotyped Class. In the latter case the numbering of tags is arbitrary, as long as type declarations are numbered lower than any declarations that use these types.

Constraints

CORBAInterface : *CORBAObjectType* (Core::Class)

All the constraints for <<*CORBAObjectType*>> apply to CORBA interfaces, as well as the following:

[1] All Attributes of a <<CORBAInterface>>-stereotyped Class must have visibility "public."

```
self.allAttributes->forAll(attrib | attrib.visibility = #public)
```

[2] All navigable opposite AssociationEnds of a <<CORBAInterface>>-stereotyped Class must have visibility "public."

```
self.navigableOppositeEnds->forAll(end | end.visibility = #public)
```

[3] A <<CORBAInterface>>-stereotyped Class tagged "isLocal" can only participate in Generalizations with other <<CORBAInterface>>-stereotyped Classes tagged "isLocal."

```
(self.generalization->forAll(
parent.isStereotyped(“CORBAInterface”) and
parent.stereotype.taggedValue->select(name = "isLocal")->size = 1))
and
(self.generalization->forAll(
child.isStereotyped("CORBAInterface") and
child.stereotype.taggedValue->select(name = "isLocal")->size = 1))
```

3.5.6.2 Notation

The notation for UML Class is used, with the stereotype keyword <<CORBAInterface>>.

Containment of data type declarations is shown using the "circle-plus" notation for UML Namespace containment.

Local interfaces are represented using the TaggedValue { isLocal = TRUE }, usually written as {local}. Non-local interfaces do not use this TaggedValue.

The UML notation for the following IDL is shown in Figure 3-17.

```
interface TestInterface {

    struct TestStruct {
        string Member1;
    };

    attribute string MyStringAttr;
    attribute TestStruct MyStructAttr;

    void MyOp1( in string str, inout TestStruct t);
    boolean MyOp2( inout TestStruct t);
};
```

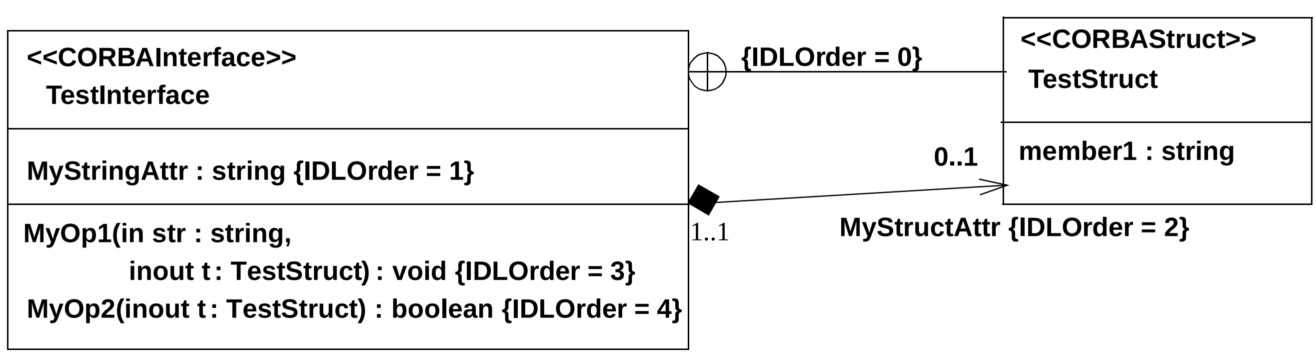

Figure 3-17 Example Interface Containing a Struct

This example shows the explicit "IDLOrder" TaggedValues on each of the Attributes, Associations, and Namespace containments for preserving the ordering given in the IDL.

3.5.7 Value Types

3.5.7.1 UML Standard Elements

Stereotypes and Tagged Values

CORBA value types are represented by a UML Class stereotyped as <<CORBAValue>>.

CORBA custom value types are represented by a UML Class stereotyped as <<CORBACustomValue>>.

Abstract value types will have their isAbstract metaattribute (from Generalizable Element) set to TRUE, and non-abstract values to FALSE. The notation reflects this in the usual UML manner by italicizing the Class name.

The support by a value type of an IDL interface type is represented by a Generalization relationship with that IDL interface's Class, which is stereotyped <<CORBAValueSupports>>.

The truncatable inheritance of one concrete value type by another is represented by a Generalization relationship between the value types that is stereotyped <<CORBATruncatable>>.

A value type factory operation is represented using a UML Operation that is stereotyped <<CORBAValueFactory>>.

Constraints

CORBAValue : *CORBAObjectType* (Core::Class)

[1] All Attributes of a <<CORBAValue>>-stereotyped Class must have visibility "public" or "private."

```
self.allAttributes->forAll(attrib | attrib.visibility = #public or
                             attrib.visibility = #private)
```

[2] All navigable opposite AssociationEnds of a <<CORBAValue>>-stereotyped Class must have visibility "public" or "private."

```
self.navigableOppositeEnds->forAll(end | end.visibility = #public or
                                    end.visibility = #private)
```

[3] A concrete <<CORBAValue>>-stereotyped Class may only specialize a single other concrete <<CORBAValue>>-stereotyped Class.

not self.isAbstract implies

```
    self.generalization
    ->select(parent.isStereokinded("CORBAValue") and
```

```
not parent.isAbstract)->size = 1
```

[4] A <<CORBAValue>>-stereotyped Class may only specialize a single <<CORBAInterface>>-stereotyped Class, and it must do so using a <<CORBAValueSupports>>-stereotyped Generalization.

```
let supportedInterface =
self.generalization->select(parent.isStereotyped("CORBAInterface")) and
let supportsGeneralization =
supportedInterface.generalization->intersection(self.generalization) in
supportedInterface->size = 1 and
supportsGeneralization.isStereotyped("CORBAValueSupports")
```

[5] A <<CORBAValue>>-stereotyped Class may only contain a single Operation stereotyped as <<CORBAValueFactory>>.

```
self.allOperations->collect(isStereotyped("CORBAValueFactory"))->size <= 1
```

Constraints

CORBACustomValue : CORBAValue (Core::Class)

As <<CORBACustomValue>> is derived from <<CORBAValue>> the constraint below applies in addition to those for <<CORBAValue>> above.

[1] A <<CORBACustomValue>>-stereotyped Class may not be truncated by a <<CORBATruncatable>>-stereotyped Generalization that specializes the Class.

```
self.generalization
->forAll(parent = self implies not isStereotyped("CORBATruncatable"))
```

CORBAValueSupports (Core::Generalization)

[1] A <<CORBAValueSupports>>-stereotyped Generalization must have a <<CORBAInterface>>-stereotyped Class as its parent and a <<CORBAValue>>-stereotyped or <<CORBACustomValue>>-stereotyped Class as its child.

```
self.parent.isStereotyped("CORBAInterface") and
self.child.isStereokinded("CORBAValue")
```

CORBATruncatable (Core::Generalization)

[1] A <<CORBATruncatable>>-stereotyped Generalization must have a concrete <<CORBAValue>>-stereotyped or <<CORBACustomValue>>-stereotyped Class as its parent and has the same restriction as to its child.

```
self.parent.isStereokinded("CORBAValue") and not self.parent.isAbstract and
self.child.isStereokinded("CORBAValue") and not self.child.isAbstract
```

CORBAValueFactory (Core::Operation)

[1] A <<CORBAValueFactory>>-stereotyped Operation can have only in parameters and has no return type.

```
self.parameter->forAll(kind = #in)
```

[2] A <<CORBAValueFactory>>-stereotypedOperation must be owned by a <<CORBAValue>>-stereotyped or <<CORBACustomValue>>-stereotyped Class.

```
self.owner.isStereokinded("CORBAValue")
```

3.5.7.2 Notation

The Class notation is used to represent CORBA value types.

For example the following IDL:

```
interface PrettyPrint {
    string print();
};

valuetype Time {
    public short hour;
    public short minute;
};

valuetype DateAndTime : Time supports PrettyPrint {
    private Date the_date;

    factory init( in short hr, in short min);
    Date get_date();
};
```

is represented in UML as shown in Figure 3-18:

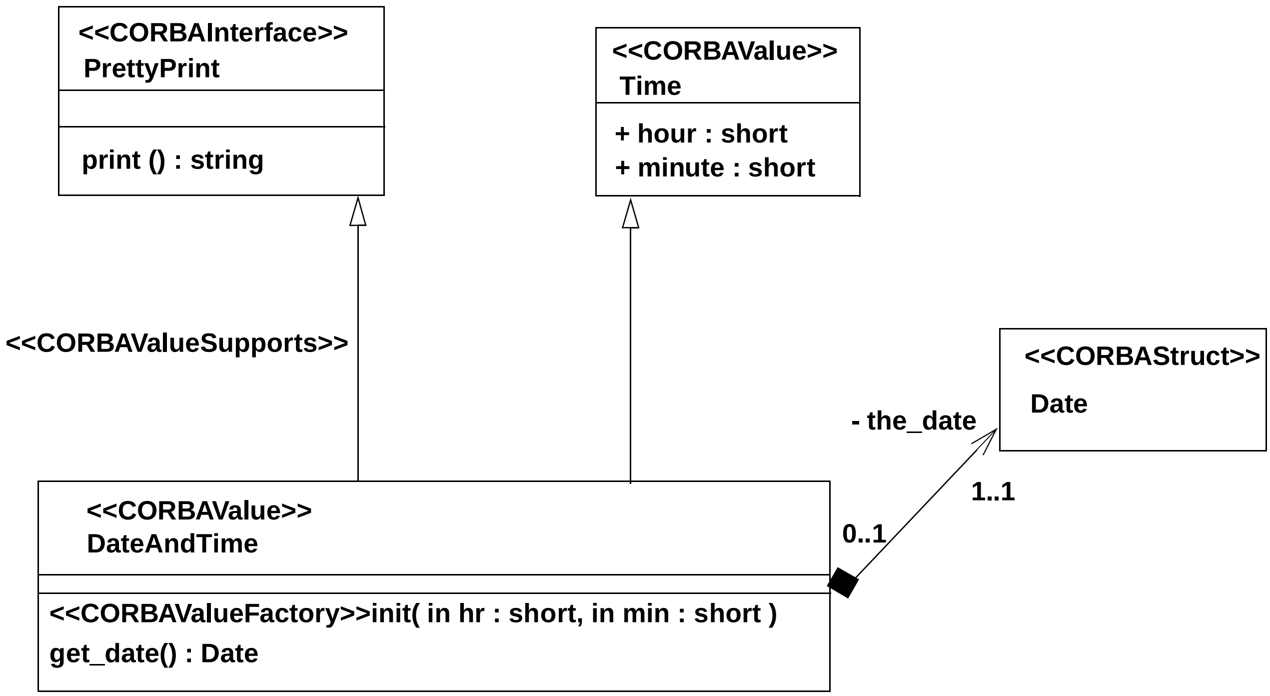

Figure 3-18 Valuetype Example

3.5.8 CORBA Wrapper Types

There are two declarations in IDL that provide existing named types with another identifier:

- typedef gives a name to an existing type (or to a new template type).
- boxed value declarations give a new name to an existing type, and allow the new type to be passed as a null parameter.

3.5.8.1 UML Standard Elements

Stereotypes and Tagged Values

Wrapper declarations are represented by the abstract stereotype <<*CORBAWrapper*>>, which specializes <<*CORBAUserDefinedType*>>. There are two concrete specializations of <<*CORBAWrapper*>>:<<CORBATypedef>> and <<CORBABoxedValue>>.

Constraints

***CORBAWrapper* : *CORBAUserDefinedType* (Core::*Classifier*)**

[1] A <<*CORBAWrapper*>>-stereotyped Classifier must participate as the child in exactly one Generalization relationship.

```
self.generalization->select(gen | gen.child = self)->size = 1
```

[2] The parent of a <<*CORBAWrapper*>>-stereotyped Classifier must be stereotyped as <<CORBAPrimitive>> or as a descendant of <<CORBAUserDefinedType>>.

```
self.generalization
->forAll(gen | gen.parent.isStereotyped("CORBAPrimitive") or
         gen.parent.isStereokinded("CORBAUserDefinedType"))
```

[3] The Generalization relationship in which a <<*CORBAWrapper*>>-stereotyped Classifier participates has the empty string as its discriminator and no powertypes.

```
self.generalization->forAll(gen | gen.discriminator = "" and
                          gen.powertype->isEmpty)
```

[4] A <<*CORBAWrapper*>>-stereotyped Classifier may not have any non-inherited features.

```
self.feature->isEmpty
```

[5] A <<*CORBAWrapper*>>-stereotyped Classifier may not participate in any Associations with navigable opposite AssociationEnds.

```
self.navigableOppositeEnds->isEmpty
```

[6] A <<*CORBAWrapper*>> can only extend a DataType or a Class.

```
self.oclIsTypeOf(DataType) or self.oclIsTypeOf(Class)
```

3.5.9 Typedef

3.5.9.1 UML Standard Elements

Typedefs in IDL serve two purposes. Firstly they rename types that already have names to provide an alias for an existing type. For example, the IDL below provides an alias “Y” for the interface named “X.”

```
interface X;
typedef X Y;
```

These typedefs are modeled by Classifiers stereotyped as <<CORBATypedef>>.

Secondly typedefs provide a type name for anonymous template types, such as sequences, arrays, and fixed point numbers. For example, the IDL below gives the name “short_array” to an anonymous array of shorts.

typedef short short_array[10];

These typedefs are modeled by Classifiers that are stereotyped as <<CORBASequence>> (see Section 3.5.19, “Sequence,” on page 3-44), <<CORBAArray>> (see Section 3.5.20, “Array,” on page 3-48) or <<CORBAFixed>> (see Section 3.5.21, “Fixed Type,” on page 3-51).

When aliasing an existing type declaration, the typedef specializes the existing ModelElements (using a UML Generalization relationship) with a new Class or Datatype being the specialization, giving the type a new name but no new features.

Stereotypes and Tagged Values

An IDL typedef aliasing a named CORBA type is represented by a UML Class or Datatype stereotyped as <<CORBATypedef>>.

Constraints

CORBATypedef : *CORBAWrapper* (Core::*Classifier*)

In addition to the constraints inherited from <<*CORBAWrapper*>>, the following also apply:

[1] The parent of a <<CORBATypedef>>-stereotyped Classifier must not be stereotyped as <<CORBAAnonymousSequence>> or <<CORBAAnonymousArray>>.

```
self.generalization
->forAll(gen |not gen.parent.isStereotyped("CORBAAnonymousSequence") and
    not gen.parent.isStereotyped("CORBAAnonymousArray"))
```

3.5.9.2 Notation

The notation for UML Class or Datatype and the notation for Generalization between Classifiers are used, with the stereotype keyword <<CORBATypedef>> attached to the Class or Datatype.

For example, the IDL definition:

typedef string Istring;
typedef Istring PropertyName;

is represented in UML as in Figure 3-19.

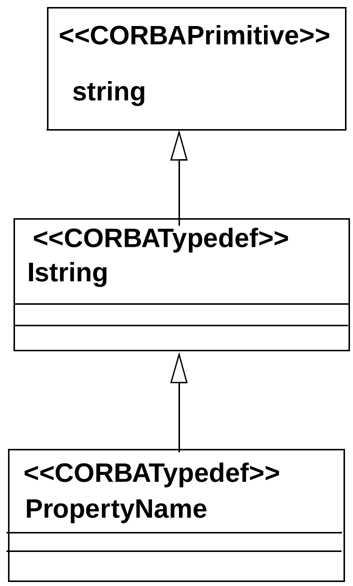

Figure 3-19 Typedef Alias Example

Examples of typedefs for anonymous types are shown in Section 3.5.19, “Sequence,” on page 3-44 and in Section 3.5.20, “Array,” on page 3-48.

3.5.10 Boxed Value Types

3.5.10.1 UML Standard Elements

Boxed values are similar to typedefs in that they provide a new name for an existing type, and change the parameter passing semantics to allow instances of the new type to be null.

When boxing an existing type declaration, the boxed value specializes the existing ModelElements (using a UML Generalization relationship) with a new Class or Datatype being the specialization, giving the type a new name, and possible null value semantics, but no new features.

Stereotypes and Tagged Values

A Boxed value type is represented by a UML Classifier stereotyped as <<CORBABoxedValue>>.

Constraints

CORBABoxedValue : *CORBAWrapper* (Core::*Classifier*)

All the constraints from <<*CORBAWrapper*>> apply to <<CORBABoxedValue>>.

3.5.10.2 Notation

The Class notation is used to represent a boxed value. The following IDL:

valuetype OptionalNameSeq sequence<string>;
valuetype OptionalStruct TestStruct;

is represented in the CORBA Profile for UML as:

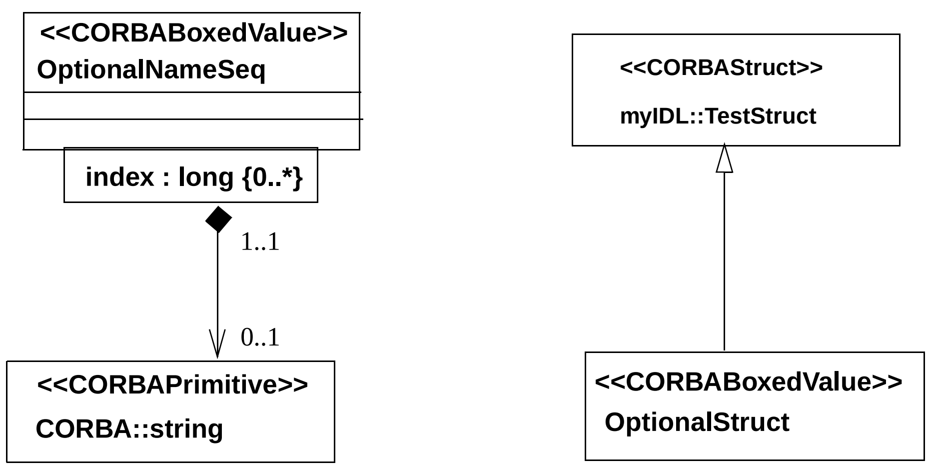

Figure 3-20 Boxed Valuetype Examples

3.5.11 Constant Declaration

3.5.11.1 UML Standard Elements

Stereotypes and Tagged Values

An IDL constant is modeled as a Stereotype “CORBAConstant” of UML Attribute, with the constant value expression represented by the Attribute’s initialValue expression.

Constants defined within the scope of an IDL interface simply become Attributes of a Class that is stereotyped as <<CORBAObjectType>>.

For constants defined within a CORBA module scope a new Stereotype “CORBAConstants” of UtilityClass is introduced. The name of the Class must be “Constants.”

Constraints

CORBAConstants (Core::Class)

[1] A <<CORBAConstants>>-stereotyped Class must be directly contained by a <<CORBAModule>>-stereotyped package.

```
self.namespace.isStereotyped("CORBAModule")
```

[2] All the features of a <<CORBAConstants>>-stereotyped Class must be <<CORBAConstant>>-stereotyped Attributes.

```
self.feature->forAll(feature | feature.oclIsTypeOf (Attribute) and
                     feature.isStereotyped ("CORBAConstant") )
```

[3] A <<CORBAConstants>>-stereotyped UtilityClass cannot participate in any Associations.

```
self.associations->isEmpty
```

Constraints

CORBAConstant (Core::Attribute)

[1] A <<CORBAConstant>>-stereotyped Attribute has the changeability “frozen” and the ownerScope “classifier.”

```
self.changeability = #frozen and self.ownerScope = #classifier
```

[2] The owner of a <<CORBAConstant>>-stereotyped Attribute must be stereotyped <<CORBAConstants>> or <<CORBAObjectType>>.

```
self.owner.isStereotyped("CORBAConstants") or
self.owner.isStereokinded("CORBAObjectType")
```

3.5.11.2 Notation

The notation for UML Attribute is used, with the stereotype name <<CORBAConstant>>.

As an example, the IDL definition:

```
module Y {
    constant short S = 3;

    interface X {
        constant long L = S + 20;
    };
};
```

is represented in UML as shown in Figure 3-21.

.

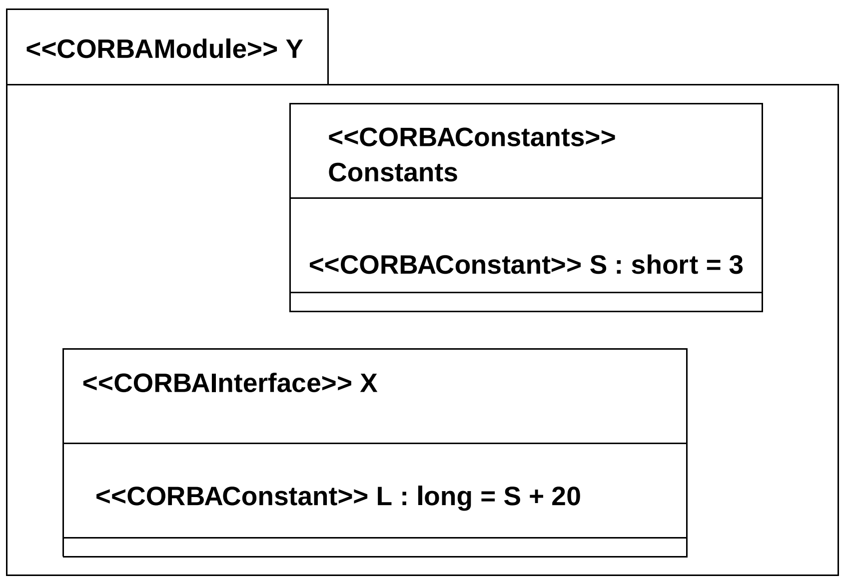

Figure 3-21 Constant Example

3.5.12 Constructed Types

This section defines the common semantics of CORBA structs, unions and exceptions, each of which share the characteristics of having ordered named elements of some CORBA type.

3.5.12.1 UML Standard Elements

Each of the IDL constructed types is represented by a stereotype of UML Classifier or one of its derived metaclasses. Specifically UML Exception is derived from Classifier, and is used to model CORBA exceptions, while UML Class is derived from Classifier and is used to model the other Constructed types.

Note – An extension of UML DataType would seem sensible here - but unfortunately DataTypes are not allowed to contain any Attributes, and Attributes are the best way to model struct/union members.

Each member of a constructed type that is of a CORBA basic type is represented as an Attribute. The name of this Attribute is the same as the identifier for the member of the constructed type.

Each member of a constructed type, which is of a user-defined IDL type, is represented as a UML Association with the stereotyped UML ModelElement representing that IDL type. In such cases, the identifier for the member of the constructed type is used as the name of the opposite AssociationEnd.

Stereotypes and Tagged Values

The virtual metamodel contains an abstract stereotype <<CORBAConstructedType>>, which is a generalization of <<CORBAStruct>>, <<CORBAUnion>>, and <<CORBAException>>. The following constraints apply to all stereotypes derived from <<*CORBAConstructedType*>>.

Each member's representation in UML must have a TaggedValue {IDLOrder = *N*}, whose integer value *N* is the position of the member's declaration in the IDL, numbered from zero upwards.

Constraints

CORBAConstructedType : CORBAStructuredType **(Core::*Classifier*)**

[1] All features of a <<*CORBAConstructedType*>>-stereotyped Classifier must be Attributes with visibility "public."

```
self.feature->forAll(feature | feature.oclIsTypeOf(Attribute) and
                     feature.visibility = #public)
```

[2] All navigable opposite AssociationEnds of a <<*CORBAConstructedType*>> must have visibility "public."

```
self.navigableOppositeEnds->forAll(end | end.visibility = #public)
```

[3] A <<*CORBAConstructedType*>>-stereotyped Classifier cannot participate in any Generalization relationships.

```
self.generalization->isEmpty and self.specialization->isEmpty
```

3.5.12.2 Notation

The notation for UML Classifier is used, with the addition of the specific stereotype keyword corresponding to the particular constructed type.

Notation examples are shown in each of the following subsections.

3.5.13 CORBAStructType

3.5.13.1 UML Standard Elements

Stereotypes and Tagged Values

The virtual metamodel contains an abstract stereotype <<CORBAStructType>>, which is a generalization of <<CORBAStruct>> and <<CORBAException>>. The following constraints apply to all stereotypes derived from <<CORBAStructType>>.

Constraints

CORBAStructType : *CORBAConstructedType* (Core::*Classifier*)

The constraints from <<*CORBAConstructedType*>> apply to all <<CORBAStructType>>-stereotyped Classifiers. In addition:

[1] All the Attributes of a <<CORBAStructType>>-stereotyped Classifier must have multiplicity 1..1.

```
self.allAttributes->forAll(multiplicity.range.lower = 1 and
                           multiplicity.range.upper = 1)
```

[2] All the navigable opposite AssociationEnds of a <<CORBAStructType>>-stereotyped Classifier must have the upper multiplicity value equal to 1.

```
self.navigableOppositeEnds->forAll(multiplicity.range.upper = 1)
```

[3] All the navigable opposite AssociationEnds of a <<CORBAStructType>>-stereotyped Classifier whose type is not a <<CORBAInterface>>-stereotyped Class must have the lower multiplicity value equal to 1.

```
self.navigableOppositeEnds->forAll(not isStereotyped("CORBAInterface")
                                   implies multiplicity.range.lower = 1)
```

3.5.13.2 Notation

See the concrete stereotypes CORBAStruct and CORBAException.

3.5.14 Struct

3.5.14.1 UML Standard Elements

Stereotypes and Tagged Values

IDL struct definitions are represented by a UML Class stereotyped as <<CORBAStruct>>.

Each basic-typed member is represented as a UML Attribute, and each user-defined-typed member is represented as an Association as defined in Section 3.5.12.1, “UML Standard Elements,” on page 3-31.

Constraints

CORBAStruct : *CORBAStructType* (Core::Class)

The constraints from <<*CORBAStructType*>> apply to all <<CORBAStruct>>-stereotyped Classes.

3.5.14.2 Notation

For example, the IDL definition:

```
struct foo {
    long length;
    PropertyName name;
    Object ref;
};
```

is represented in UML as shown in Figure 3-22.

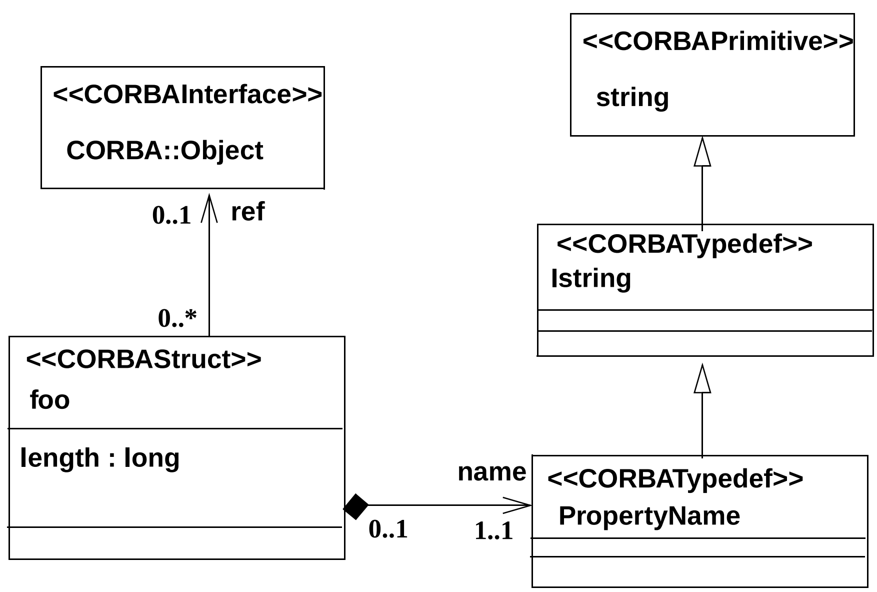

Figure 3-22 Struct Example

The following IDL:

```
struct A {
    struct B {
        short k;
        long j;
    } p;
    string q;
};
```

is represented in UML as in Figure 3-23.

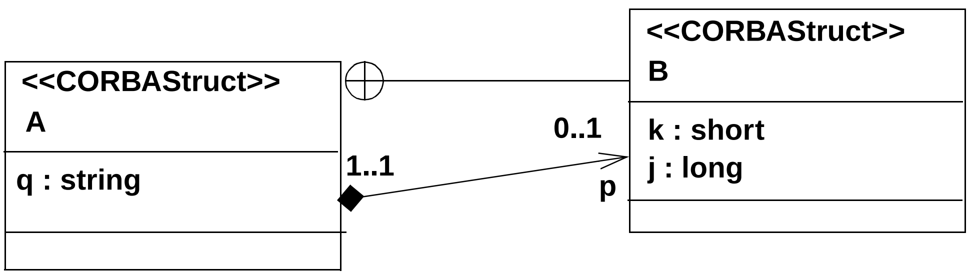

Figure 3-23 Nested Struct Example

Note – Whether the multiplicity on the AssociationEnd named “p” is 0..1 or 1..1 is the modelers choice.

3.5.15 Discriminated Union

3.5.15.1 UML Standard Elements

Stereotypes and Tagged Values

IDL union definitions are represented by a UML Class stereotyped as <<CORBAUnion>>.

The discriminator type is represented as an additional Attribute or Association (according to the rules given in Section 3.5.12.1, “UML Standard Elements,” on page 3-31) of the <<CORBAUnion>>-stereotyped Class, which is stereotyped as a <<switch>> or <<switchEnd>> respectively.

Each member of the IDL union is represented as a UML Attribute or Association, according to the rules given in Section 3.5.12.1, “UML Standard Elements,” on page 3-31.

Each member has a TaggedValue {IDLOrder = *N*} attached to it, where *N* is a number from zero upwards corresponding to the order in which the members are declared.

Each member has a TaggedValue {Case = *LabelName*} attached with its *LabelName* value being the case label for this member in the union declaration. For union declarations in which there is a default case, the value of *LabelName* for the default member will be the string “default.”

Constraints

CORBAUnion : *CORBAConstructedType* (Core::Class)

All of the constraints on <<*CORBAConstructedType*>> apply to unions. In addition the following constraints apply:

[1] Either exactly one of the Attributes or exactly one of the navigable opposite AssociationEnds of a <<CORBAUnion>>-stereotyped Class (but not both) must be stereotyped as <<switch>> (in the case of an Attribute) or <<switchEnd>> (in the case of AssociationEnd).

```
self.allAttributes->select(attrib | attrib.type.isStereotyped("switch") )-
>size = 1 xor
self.navigableOppositeEnds->select(end |
end.type.isStereotyped("swutchEnd") )->size = 1
```

[2] The Attribute or AssociationEnd that represents the switch of the IDL union represented by the <<CORBAUnion>>-stereotyped Class must have multiplicity 1..1.

```
let switch =
                  self.allAttributes->select(attrib |
attrib.type.isStereotyped("switch"))->union(
     self.navigableOppositeEnds->select(end |
end.type.isStereotyped("switchEnd"))) in


switch.oclIsTypeOf(Attribute) implies
                 (switch.multiplicity.range.lower = 1 and
                 switch.multiplicity.range.upper = 1)
and
switch.oclIsTypeOf(AssociationEnd) implies
                 (switch.multiplicity.range.lower = 1 and
                 switch.multiplicity.range.upper = 1)
```

[3] With the exception of the element representing the switch, every Attribute and navigable opposite AssociationEnd of a <<CORBAUnion>>-stereotyped Class must have the multiplicity 0..1 and a tagged value “case.”

```
(self.allAttributes
->forAll(attrib | not attrib.type.isStereotyped("switch")
    implies attrib.multiplicity.range.lower = 0 and
             attrib.multiplicity.range.upper = 1))
and
```

```
(self.navigableOppositeEnds
->forAll(end | not end.type.isStereotyped("switchEnd")
    implies end.multiplicity.range.lower = 0 and
            end.multiplicity.range.upper = 1))
```

3.5.15.2 Notation

The UML Class notation is used. For example, the IDL definition:

```
enum Contents {
    INTEGER_CL;
    FLOAT_CL;
    DOUBLE_CL;
    COMPLEX_CL;
    STRUCTURED_CL;
};

union Reading switch (Contents) {
    case INTEGER_CL:
            long a_long;
    case FLOAT_CL:
    case DOUBLE_CL:
            double a_double;
    default:    any an_any;
};

union ValOpt switch (boolean) {
    case TRUE: PropertyValue pv;
};
```

is represented in UML as in Figure 3-24.

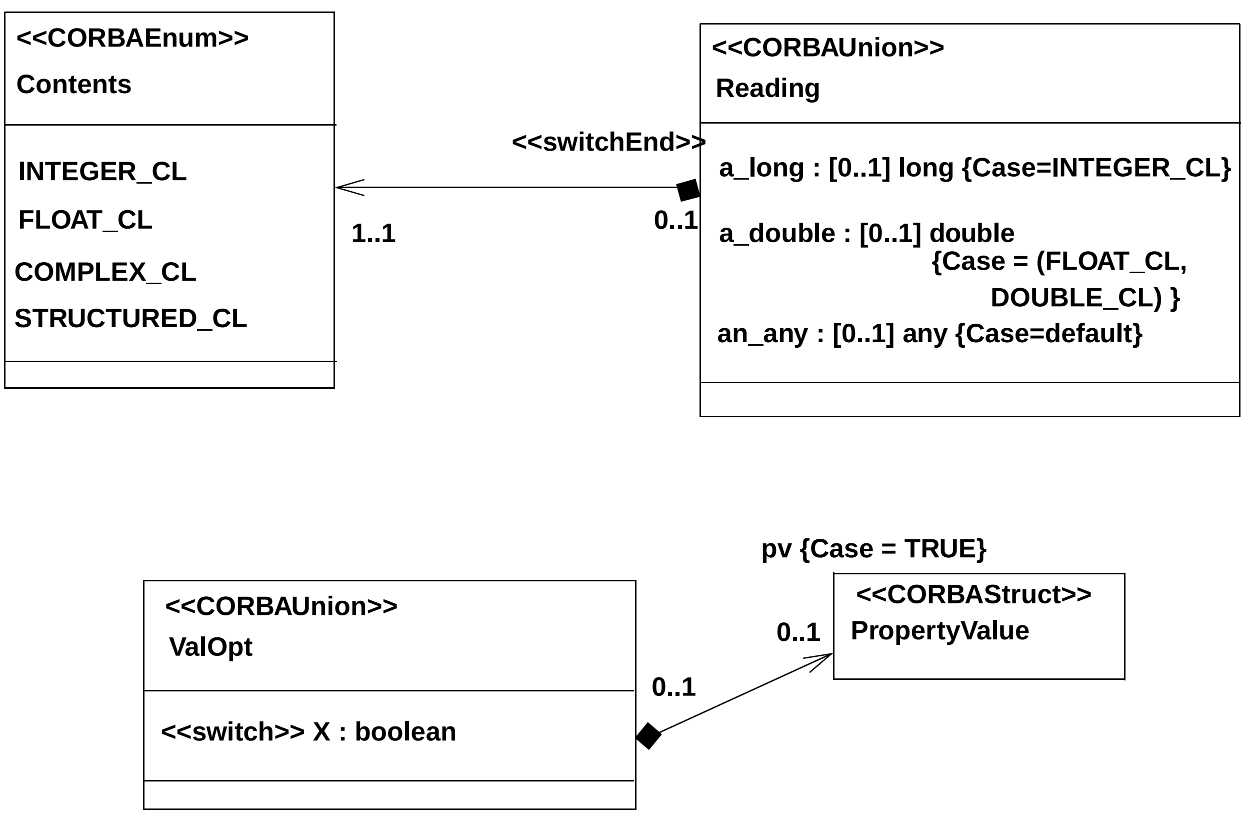

Figure 3-24 Union Example

3.5.16 Enum

3.5.16.1 UML Standard Elements

Stereotypes and Tagged Values

IDL enum definitions are represented by UML Classes stereotyped as <<CORBAEnum>>.

Each element of the enum type is represented as an Attribute of the stereotyped UML Class, with the same name as the enum element.

Constraints

CORBAEnum : *CORBAConstructedType* (Core::Class)

All constraints that apply to <<*CORBAConstructedType*>> apply to enums. In addition:

[1] All the Attributes of a <<CORBAEnum>>-stereotyped Class must have multiplicity 1..1, type CORBA::short and an initialValue equal to the value of its IDLOrder tag.

```
self.allAttributes
->forAll(attrib |attrib.multiplicity.range.lower = 1 and
            attrib.multiplicity.range.upper = 1 and
            attrib.initialValue.body = attrib.taggedValues
            ->select(tag | tag.name = "IDLOrder").value)
```

[2] A <<CORBAEnum>>-stereotyped Class may not participate in any Association that has navigable opposite AssociationEnds.

```
navigableOppositeEnds->isEmpty
```

3.5.16.2 Notation

The usual Class and Attribute notation is used. An example enum is represented in UML in Figure 3-25 on page 3-41.

The type and initial numeric values of the Attributes representing enum elements may be omitted in the notation, as the type is always short, and the initialValue can be deduced from the ordering of the Attributes.

3.5.17 Exception

3.5.17.1 UML Standard Elements

CORBA Exceptions are modeled using stereotypes of UML Exceptions. The members of the CORBA exception are represented exactly the same as those of CORBA struct.

In the UML metamodel, Exception is a subtype of Signal, which is a subtype of Classifier.

Note – In CORBA 2.4 attributes may not raise exceptions.

Stereotypes and Tagged Values

CORBA exceptions are represented by UML Exceptions (from Common Behavior), stereotyped <<CORBAException>>.

The TaggedValue { raises = (*exception-name*, *exception-name*, ...) } is defined for Operations in Section 3.5.22, “Operation,” on page 3-53 to describe which operations raise which exceptions.

Constraints

CORBAException : *CORBAStructType* (CommonBehavior::Exception)

All the constraints for <<*CORBAStructType*>>-stereotyped Classifiers apply to <<CORBAException>>-stereotyped Exceptions. The following constraint also applies.

[1] A <<CORBAException>>-stereotyped Exception cannot be the type of a navigable AssociationEnd.

```
self.allEnds->forAll(end | end.type = self implies not end.isNavigable)
```

3.5.17.2 Notation

The notation for UML Class is used to represent exceptions.

Examples

The following IDL:

```
struct AdminLimit {
    CosNotification::PropertyName name;
    CosNotification::PropertyValue val;
};

exception AdminLimitExceeded { AdminLimit admin_property_err; };
```

is represented in UML as in Figure 3-25.

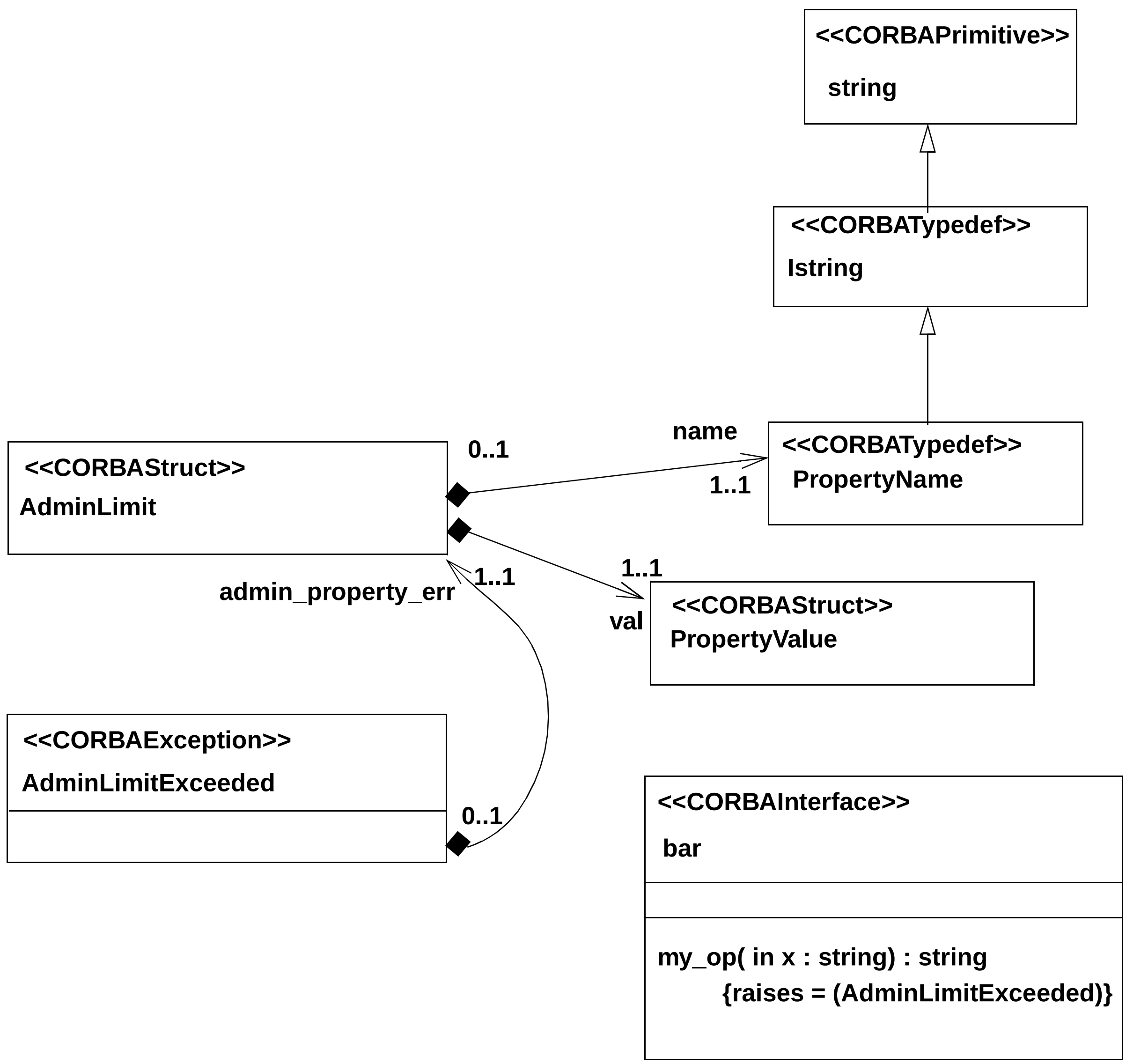

Figure 3-25 Operation Raising Exception Example

The following IDL:

```
interface Tex {

    exception Badness2000 { string err_msg };

    void process_token ( in string tok)
        raises (Badness2000);
};
```

is represented in UML as in Figure 3-26.

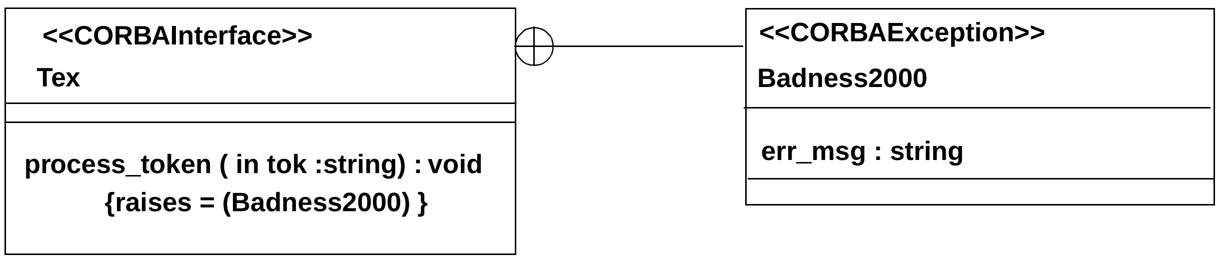

Figure 3-26 Exception Containment and Raising

3.5.18 Indexed Types

The CORBA indexed types are sequences and arrays. These have similar modeling characteristics, and so we define their commonalities using the abstract stereotype <<*CORBAIndexedType*>>. All indexed types are modeled as Classes that have a qualified Association to the element type. The qualifier represents the index. We model indexed types in two ways, depending on whether or not the element type is an interface type.

Case 1: Where an interface type is being indexed:

- The opposite end multiplicity on the qualified association defaults to 0..1.
- The near end multiplicity on the qualified association defaults to 0..*
- The near end AggregationKind defaults to “none.”

Case 2: Where a non-interface type is being indexed:

- The opposite end multiplicity on the qualified association is 1..1.
- The near end multiplicity on the qualified association defaults to 0..1.
- The near end AggregationKind is “composite.”

3.5.18.1 UML Standard Elements

Stereotypes and Tagged Values

The abstract stereotype <<*CORBAIndexedType*>> specializes the abstract stereotype <<*CORBAStructuredType*>>, and all its concrete derived stereotypes are of UML Class.

<<*CORBAIndexedType*>> is specialized to the concrete stereotypes <<CORBASequence>>, <<CORBAAnonymousSequence>>, <<CORBAArray>>, and <<CORBAAnonymousArray>>.

Constraints

***CORBAIndexedType : CORBAUserDefinedType* (Core::Class)**

[1] A <<*CORBAIndexedType*>>-stereotyped Class has no features.

```
self.features->isEmpty
```

[2] A <<*CORBAIndexedType*>>-stereotyped Class participates in exactly one Association that has a navigable opposite AssociationEnd.

```
self.navigableOppositeEnds->size = 1
```

[3] The single navigable opposite AssociationEnd of a <<*CORBAIndexedType*>>-stereotyped Class must have the visibility “public.”

```
self.navigableOppositeEnds->forAll(end | end.visibility = #public)
```

[4] There is exactly one Association in which a <<*CORBAIndexedType*>> participates as the type of the near, non-navigable AssociationEnd.

```
self.nonNavigableNearEnds->size = 1
```

[4] All qualifiers of the single non-navigable near AssociationEnd of a <<*CORBAIndexedType*>> must have multiplicity 1..1, type CORBA::long, and a constraint of the form "{0..n}" or "{0..*}" (where n is a non-negative integer).

```
self.nonNavigableNearEnds->forAll(end | end.qualifier->forAll
(qualifier | qualifier.multiplicity.range.lower = 1 and
    qualifier.multiplicity.range.upper = 1 and
    qualifier.type.name = "CORBA::long" and
    qualifier.constraint->exists(constraint |
        constraintSubstring(constraint,1,3) = "0.." and
        (constraintUpperValue = "*" or
        constraintUpperValue >= 1))))
```

[5] If the type being indexed is a <<CORBAInterface>>-stereotyped Class, then the opposite AssociationEnd of the one qualified Association in which the <<*CORBAIndexedType*>>-stereotyped Class participates has multiplicity lower bound of at most 1 and multiplicity upper bound of exactly 1.

```
self.navigableOppositeEnd.type.isStereotyped("CORBAInterface")
implies
    self.navigableOppositeEnd.multiplicity.range.lower <= 1 and
    self.navigableOppositeEnd.multiplicity.range.upper = 1
```

[6] If the type being indexed is not a <<CORBAInterface>>-stereotyped Class, then

- the opposite AssociationEnd of the one qualified Association in which the <<*CORBAIndexedType*>>-stereotyped Class participates has multiplicity 1..1, and
- the near AssociationEnd of the qualified Association has multiplicity lower bound of at most 1 and upper bound of exactly 1.

```
not
self.qualifiedAssociationOppositeEnd.type.isStereotyped("CORBAInterface")
implies
    (self.navigableOppositeEnd.multiplicity.range.lower = 1 and
    self.navigableOppositeEnd.multiplicity.range.upper = 1)
    and
    (self.navigableNearEnd.multiplicity.range.lower <= 1 and
    self.navigableNearEnd.multiplicity.range.upper =1)
```

[7] A <<*CORBAIndexedType*>>-stereotyped Class cannot have any ownedElements.

```
self.ownedElements->isEmpty
```

[8] A <<*CORBAIndexedType*>>-stereotyped Class cannot participate in any Generalization relationships.

```
self.generalization->isEmpty and self.specialization->isEmpty
```

OCL Convenience Operations

[1] and [2] deleted

[3] The operation **constraintAsString** returns a substring of the Constraint body expression, where *lower* is the lower bound of the substring and *upper* is the upper bound of the substring.

```
constraintSubstring :
(constraint : Constraint, lower : Integer, upper : Integer);

constraintSubstring =
constraint.body.body.substring(lower,upper) --sic!
```

[4] The operation **constraintUpperValue** returns the portion of the Constraint body expression following “0..”

```
constraintUpperValue : (constraint : Constraint);

constraintUpperValue =
constraintSubString(constraint,4,constraint.body.body.size) --sic!
```

3.5.19 Sequence

CORBA Sequences are IDL template types that take a CORBA type as their element parameter, and optionally an integer as an upper bound specification. Sequences are anonymous, and can either be named by a typedef, or by the member name of a constructed type.

Sequences are modeled by Classes that participate in a qualified Association with the Classifier representing the element type of the sequence. The qualifier Attribute on the Association represents the index of the sequence, and the (optional) upper bound of the sequence is modeled as a constraint on that index.

Sequences that are declared as the type-declarator of a typedef are given the name of that typedef and the stereotype <<CORBASequence>>. Sequences that are anonymous (declared in some context where they don't have a type name, such as a struct member type) are given the stereotype <<CORBAAnonymousSequence>>.

3.5.19.1 UML Standard Elements

Stereotypes and Tagged Values

An IDL sequence that is declared as the type-declarator of a typedef is represented as a UML Class stereotyped as <<CORBASequence>>. The name of the Class will be the name of the typedef.

An IDL sequence that is declared in any other context is represented by a Class stereotyped as <<CORBAAnonymousSequence>>.

Constraints

CORBASequence : *CORBAIndexedType* (Core::Class)

These constraints are in addition to those defined for all CORBA indexed types in Section 3.5.18, "Indexed Types," on page 3-42.

[1] The single non-navigable near AssociationEnd of a <<CORBASequence>>-stereotyped Class must have a single qualifier with the name "index."

```
self.nonNavigableNearEnds->size = 1 and
self.nonNavigableNearEnds
->forAll(end |end.qualifier->size = 1 and
        end.qualifier->forAll(qualifier | qualifier.name = "index")
```

[2a] The single navigable opposite AssociationEnd of a <<CORBASequence>>-stereotyped Class must have multiplicity 1..1 if it cannot be a null in CORBA; that is, unless it is an object type or a boxed value type.

[2b] The single navigable opposite AssociationEnd of a <<CORBASequence>>-stereotyped Class must have multiplicity 0..1 if it is a boxed value type or object type.

[3] A <<*CORBAIndexedType*>>-stereotyped Class cannot have any ownedElements.

```
self.ownedElements->isEmpty
```

Constraints

CORBAAnonymousSequence : CORBASequence (Core::Class)

In the virtual metamodel <<CORBAAnonymousSequence>> is a derived Stereotype of <<CORBASequence>>; therefore, the following constraint is in addition to those for <<CORBASequence>> above.

[1] A <<CORBAAnonymousSequence>>-stereotyped Class must have exactly one navigable opposite AssociationEnd whose multiplicity is 1..1.

```
navigableOppositeEnds->size = 1 and
navigableOppositeEnds
->forAll(end | end.multiplicity.range.lower = 1 and
         end.multiplicity.range.upper = 1)
```

3.5.19.2 Notation

The UML Class and qualified Association notation is used. For example, the IDL definition:

```
typedef sequence<short> foo;

struct bar {
    long val;
    sequence <short, 4> my_shorts;
};
```

is represented in UML as in Figure 3-27.

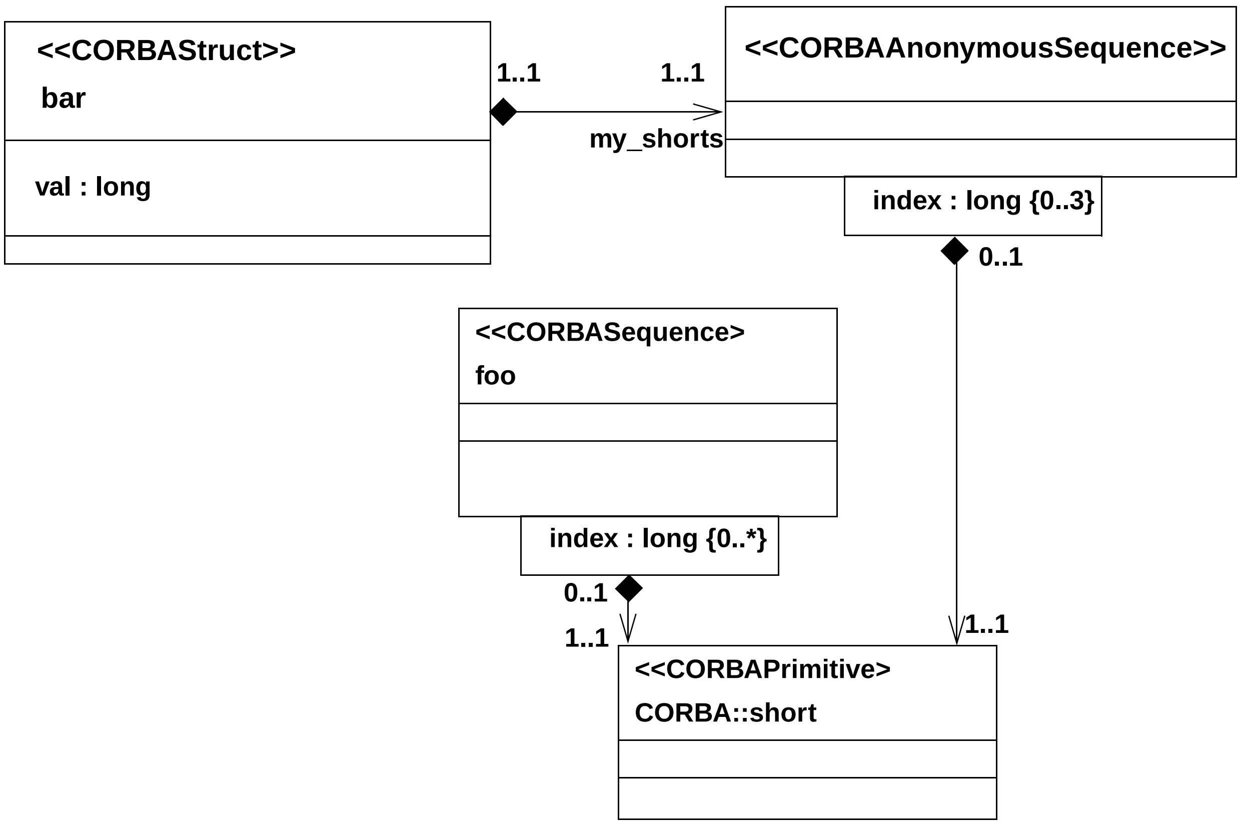

Figure 3-27 Sequence Example

The following IDL, featuring an anonymous sequence as the type of another sequence:

typedef sequence < sequence < string > > string_matrix;

is represented in UML as in Figure 3-28.

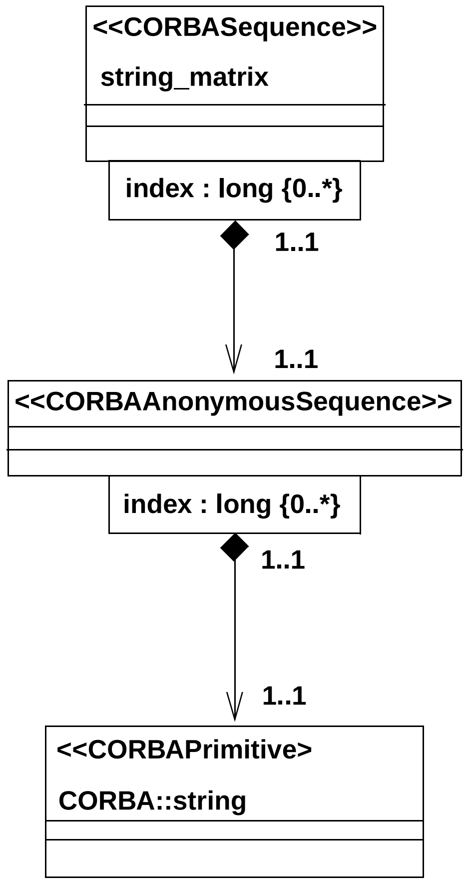

Figure 3-28 Nested Sequence Example

3.5.20 Array

CORBA Arrays are IDL indexed types that take a CORBA type as their element type, and have at least one integer as the size of the zeroth dimension of the array. Additional array dimensions are specified by additional integers. Arrays are anonymous, and can either be named by a typedef, or by the member name of a constructed type.

Arrays are modeled by Classes that participate in a qualified Association with the Classifier representing the element type of the array. The qualifier on the Association contains Attributes representing each dimension of the array. A constraint on each qualifier Attribute represented specifies the size of the array dimension. The type of all qualifier Attributes is CORBA::long, and constraints on the Attributes limit these to have a value between zero and size-1.

Arrays that are declared as the type-declarator of a typedef are given the name of that typedef and the stereotype <<CORBAArray>>. Arrays that are anonymous (declared in some context where they don't have a type name, such as a struct member type) are given the stereotype <<CORBAAnonymousArray>>.

3.5.20.1 UML Standard Elements

Stereotypes and Tagged Values

An IDL array that is declared as the type-declarator of a typedef is represented as a UML Class stereotyped as <<CORBAArray>>. The name of the Class will be the name of the typedef.

An IDL array that is declared in any other context is represented by a Class stereotyped as <<CORBAAnonymousArray>>.

Constraints

CORBAArray : CORBAIndexedType (Core::Class)

[1] The single non-navigable near AssociationEnd of a <<CORBAArray>>-stereotyped Class must have one or more qualifiers with the names "index<i>," where the <i> are contiguous integers starting from 0.

```
let dimensions = self.nonNavigableNearEnd.
qualifier->collect(name.substring(6,name.size)) in
self.nonNavigableNearEnd.qualifier
->forAll(name.substring(1,5) = "index") and

dimensions->isUnique(n | n) and
dimensions->forAll(dim | dim >= 0 and dim <= dimensions.size)
```

[2] The single navigable opposite AssociationEnd of a <<CORBAArray>>-stereotyped Class must have multiplicity 1..1.

```
navigableOppositeEnds
->forAll(end | end.multiplicity.range.lower = 1 and
                 end.multiplicity.range.upper = 1)
```

CORBAAnonymousArray : CORBAArray (Core::Class)

As <<CORBAAnonymousArray>> specializes <<CORBAArray>> in the virtual metamodel, the following constraints apply in addition to those for arrays above.

[1] A <<CORBAAnonymousArray>>-stereotyped Class must have exactly one navigable opposite AssociationEnd whose multiplicity is 1..1.

```
navigableOppositeEnds->size = 1 and
```

```
navigableOppositeEnds
->forAll(end | end.multiplicity.range.lower = 1 and
              end.multiplicity.range.upper = 1)
```

3.5.20.2 Notation

The UML notation for Classes and for qualified Associations is used to represent arrays.

Examples

The following IDL is represented in UML as in Figure 3-29.

typedef short short_arr[4];
typedef my_struct my_struct_arr[5][10];

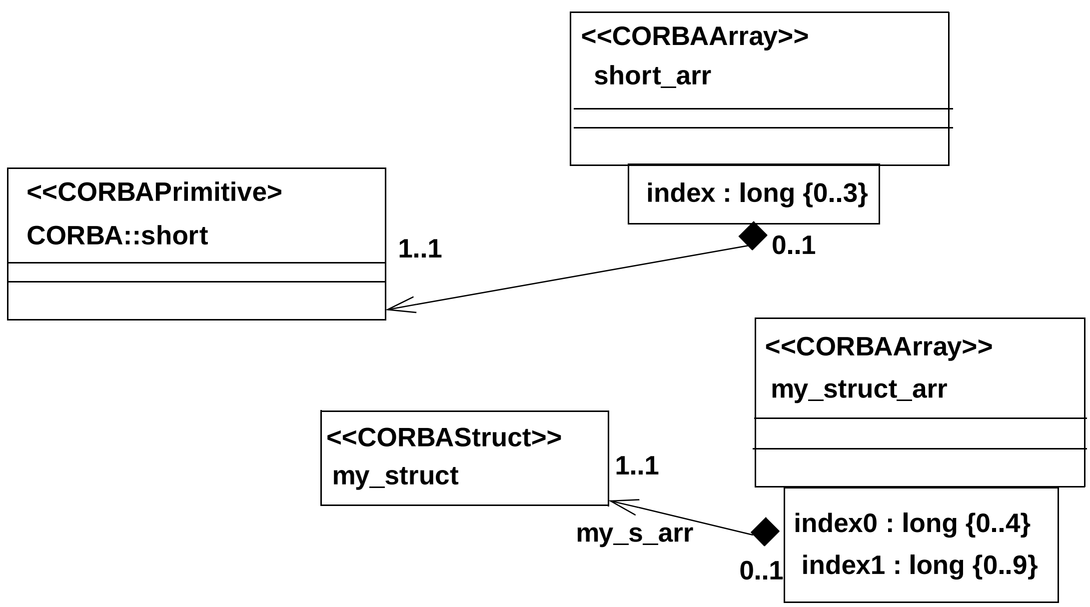

Figure 3-29 Typedefed Array Example

Note – The AssociationEnd name my_s_arr above does not affect the IDL. Modelers may choose to provide meaningful names for AssociationEnds of Associations between Classes representing sequences and arrays and their element types, or to provide empty string names, or to suppress the names in their notation.

The following IDL is represented in UML as in Figure 3-30.

```
struct boom {
    string zoom[4];
    my_struct loom[2][2][2];
};
```

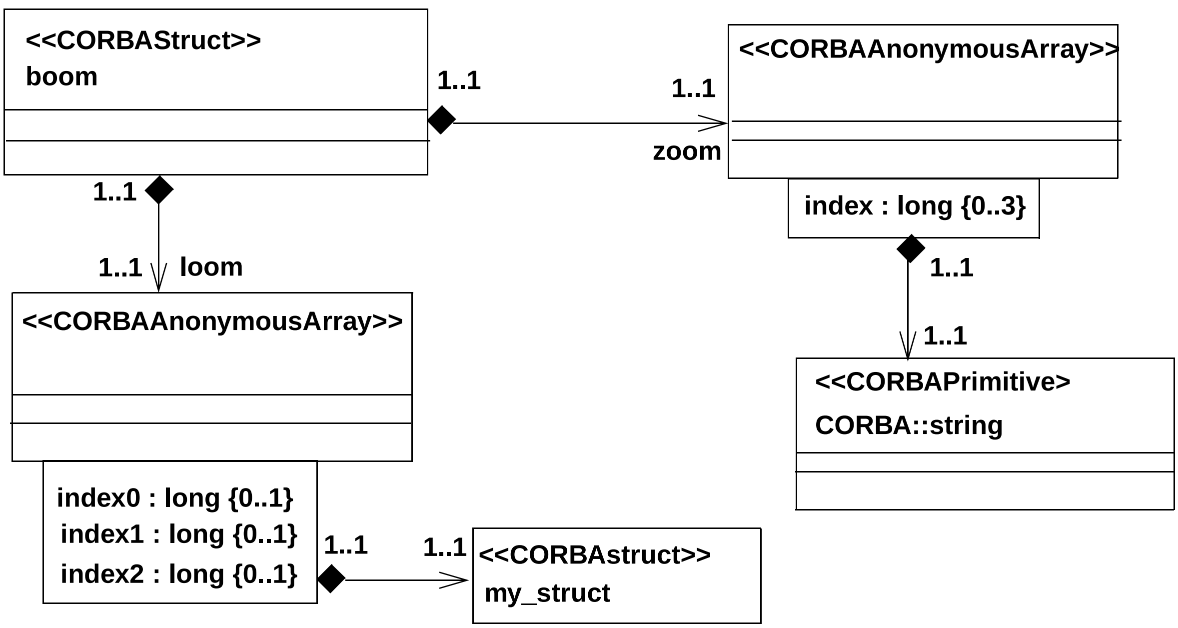

Figure 3-30 Anonymous Array Example

3.5.21 Fixed Type

Fixed types are modeled as Template Classes with two Parameters representing the digits and scale of a fixed point decimal number. Specific uses of fixed point declarations will instantiate (bind) the template with specific parameters.

When a typedef gives the fixed point type instantiation a name, the resulting Class is stereotyped as <<CORBAFixed>>.

3.5.21.1 UML Standard Elements

Stereotypes and Tagged Values

Fixed point number types are modeled by UML Classes that instantiate the Template CORBA::fixed. Any IDL fixed declaration, which is the declarator type of a typedef is given the stereotype <<CORBAFixed>>. Fixed declarations in other contexts are not stereotyped.

Constraints

CORBAFixed : *CORBAStructuredType* (Core::Class)

All constraints that apply to <<*CORBAStructuredType*>> apply to IDL fixed instances.

Common ModelElements

The Template Class “fixed” is contained in the CORBA package.

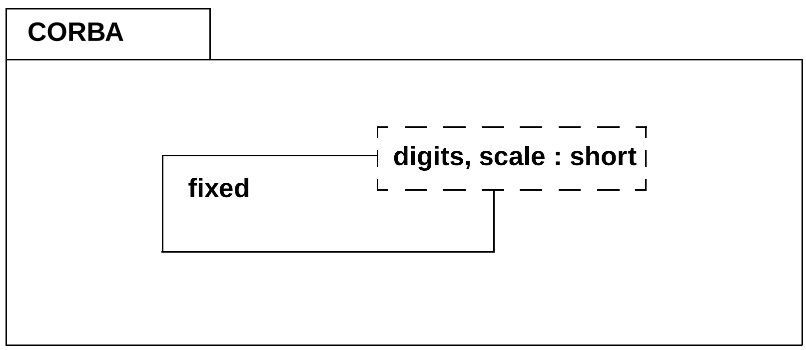

Figure 3-31 Fixed Template Specification

3.5.21.2 Notation

Uses of anonymous fixed types as members of constructed types use the inline notation:

member-name **: fixed <*digits*, *scale*>**

in the same way as IDL.

Typedefed fixed types bind the template parameters using the <<bind>> stereotype. These Classes are stereotyped as <<CORBAFixed>>.

The following IDL would be represented as shown in Figure 3-32.

```
typedef fixed <10, 2> bar;

struct baz {
    fixed <8, 4> high_scale;
    fixed <8, 2> low_scale;
};
```

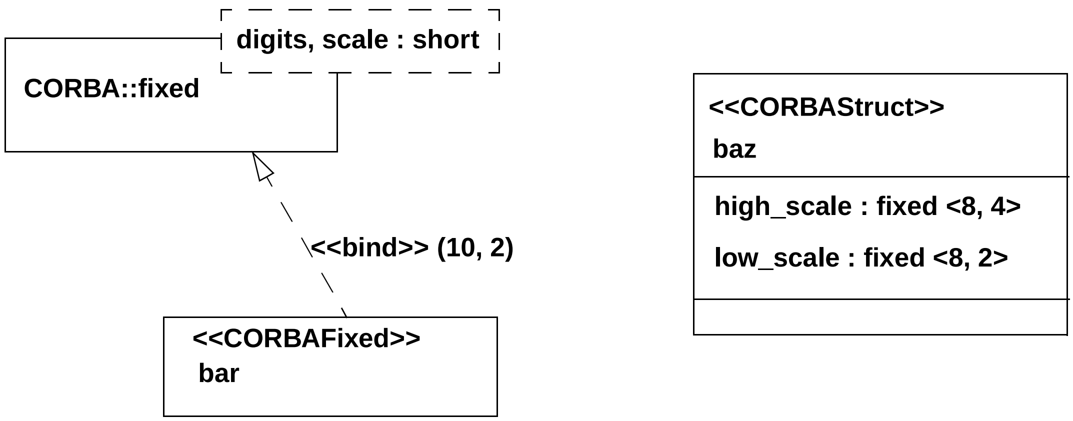

Figure 3-32 Fixed Example

3.5.22 Operation

3.5.22.1 UML Standard Elements

An IDL operation is represented as a UML Operation of the <<*CORBAObjectType*>>-stereotyped UML Class that represents the containing CORBA interface.

The names and types of the operation parameters are mapped directly to the UML Operation Parameter names and types, regardless of whether the types are IDL basic types or user defined IDL types. The IDL Parameter directional attributes “in,” “out,” and “inout” are represented using the equivalent UML Parameter kind metaattribute.

Stereotypes and Tagged Values

CORBA oneway operations are modeled by UML Operations stereotyped as <<oneway>>.

Context expressions on CORBA operations are modeled using a TaggedValue {Context = (*ctx_expr*, *ctx_expr*, ...) }.

Raises clauses on IDL operations are modeled using a TaggedValue { raises = (*exception-name*, *exception-name*, ...) }. Where the names of exceptions raised are given in UML scope identifier notation.

Constraints

oneway (Core::Operation)

[1] A <<oneway>>-stereotyped Operation may not have any out or inout Parameters and must have a return Parameter of type CORBA::void.

```
self.parameter
->forAll(param | param.kind <> #out and
              param.kind <> #inout) and
self.parameter->select(param | param.kind = #return and
              param.type.name = “CORBA::void”)->size = 1)
```

[2] A <<oneway>>-stereotyped Operation must be owned by a <<CORBAInterface>>-stereotyped Class.

```
self.owner.isStereotyped("CORBAInterface")
```

3.5.22.2 Notation

The notation for UML Operation in a UML Classifier is given by the UML 1.3 Notation Guide as:

stereotype-keyword visibility name ‘(‘ *parameter-list* ‘)’ *: return-type-expression* ‘{‘ *property-string* ‘}’

The optional notation for *visibility* is not required because it will always be stored in the model as “public.”

For oneway operations the *stereotype-keyword* <<oneway>> is placed before the name of the operation.

The Operation *name* is the same as the IDL operation name.

The parameter-list is defined below.

The *return-type-expression* will use UML double-colon-separated scoped naming conventions when it refers to types defined outside the module naming scope within which this IDL operation is declared.

Parameter List

The standard UML syntax used to represent parameter-lists is a comma-separated list of parameters, each of which are shown as follows:

kind name : type-expression = default-value

In UML the *kind* label (**in, out, inout**) is optional - but for this Profile it is mandated that it be represented, even for the default **in** case.

The *type-expression* will use UML double-colon-separated scoped naming conventions when it refers to types defined outside the module naming scope within which this IDL operation is declared.

The *default-value* specification is optional, and may be used by a modeler to provide instruction to implementers, but it is not representative of any IDL semantics.

Exception Raising

The specification that a particular IDL exception may be raised by an IDL operation is denoted by a TaggedValue “raises” defined in Section 3.5.22, “Operation,” on page 3-53.

Example

The following IDL of an interface with several operations from the Notification Service is shown using the CORBA Profile in Figure 3-33 on page 3-56.

```
interface SequencePullSupplier : NotifySubscribe {

    CosNotification::EventBatch pull_structured_events(
            in long max_number )
        raises(CosEventComm::Disconnected);

    CosNotification::EventBatch try_pull_structured_events(
            in long max_number,
            out boolean has_event)
        raises(CosEventComm::Disconnected);

    void disconnect_sequence_pull_supplier();

}; // SequencePullSupplier
```

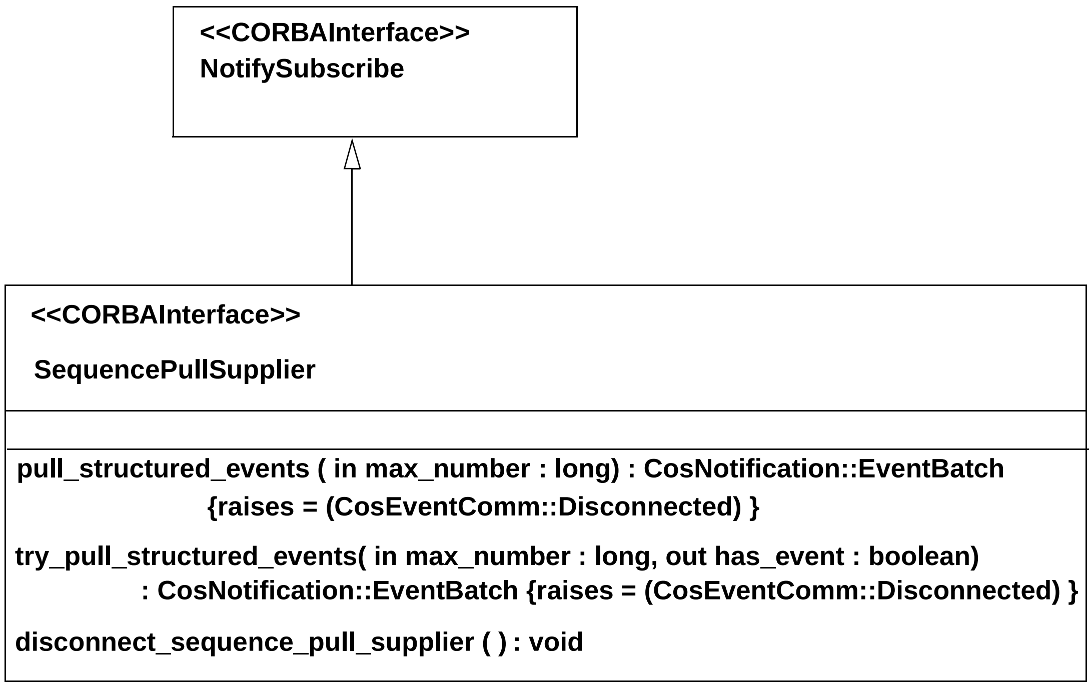

Figure 3-33 Example IDL Interface with Operations

The following IDL interface containing oneway operations and context expressions is depicted in Figure 3-34.

```
interface OutOfDate {
    oneway void signalExpiry( in Date expiry_date );
    boolean validContext( ) context ( “ptr*”, “rtn*” );
}
```

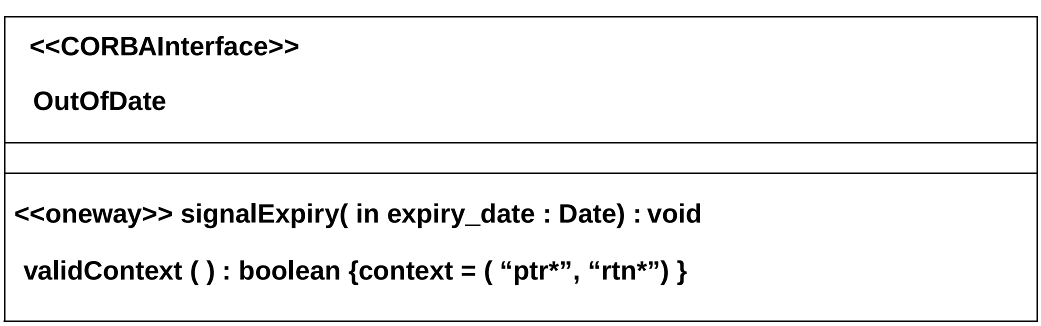

Figure 3-34 Example IDL Interface with Oneway

3.5.23 Attribute

3.5.23.1 UML Standard Elements

An IDL attribute definition whose type is CORBA basic types in an IDL interface or valuetype definition is represented as:

- A UML Attribute of a <<CORBAInterface>>-stereotyped Class corresponding to the IDL interface that the attribute is defined in. The identifier of the IDL attribute is used as the name of the UML Attribute in the <<CORBAInterface>> stereotyped Class.

An IDL attribute whose type is a user-defined data type is represented as:

- A UML Association between the <<CORBAInterface>>-stereotyped Class and the UML stereotype that represents the user-defined IDL type of the IDL attribute. The identifier of the IDL attribute is used as the role name for the user-defined-type AssociationEnd of this Association.

Stereotypes and Tagged Values

CORBA readonly attributes of basic types are represented by UML Attributes stereotyped as <<readonly>>.

CORBA readonly attributes of user-defined types are represented by UML Associations between the interface or value type defining the attribute, and the Classifier representing the attribute's type. The far (type) AssociationEnd of this Association will be stereotyped <<readonlyEnd>>.

The specification that a particular exception type may be raised by an IDL attribute's access (get) or modification (set) is denoted by a pair of Tagged Values:

{ getRaises = (*exception-name*, *exception-name*, ...) }

{ setRaises = (*exception-name*, *exception-name*,...) }

Where the exception-name is the name of the <<CORBAException>>-stereotyped exception raised by the attribute's accessor and modifier.

Constraints

readonly (Core::Attribute)

[1] A <<readonly>>-stereotyped Attribute cannot raise a modify exception.

```
not self.taggedValue->exists(tag | tag.name = "setRaises")
```

readonlyEnd (AssociationEnd)

[1] A <<readonlyEnd>>--stereotyped AssociationEnd cannot raise a modify exception.

```
not self.taggedValue->exists(tag | tag.name = "setRaises")
```

3.5.23.2 Notation

Basic Typed Attributes

IDL attributes whose type is a CORBA basic type are shown using the inline form for UML Attributes, which is given in the UML 1.3 Notation Guide is as follows:

stereotype-keyword visibility name '[' *multiplicity* ']' ':' *type-expression* '=' *initial-value* '{' *property-string* '}'

Readonly attributes will have the *stereotype-keyword* <<readonly>> at the beginning of the attribute.

The *visibility* notation is mandatory for Attributes of value types. For Attributes of CORBA interfaces the visibility is always "public," but thus may be suppressed in the notation.

The *name* shown is the same as the name of the IDL attribute name.

The optional square-bracketed *multiplicity* is always 1..1 and is not shown in the notation.

The *type-expression* will use CORBA double-colon-separated scoped naming conventions when it refers to types defined outside the module naming scope within which this IDL operation is declared.

The *initial-value* specification is optional, and may be used by a modeler to provide instruction to implementers, but it is not representative of any IDL semantics.

IDL readonly attributes must show the TaggedValue {readonly} after the Attribute.

User-defined Typed Attributes

The notation for UML Association is used to denote IDL attributes that have user-defined IDL types.

Readonly IDL attributes have the stereotype <<readonlyEnd>> attached to the AssociationEnd corresponding to the attribute's type.

Unless the IDL attribute is an IDL object reference the Association is modeled as a strong aggregation, also known as a composite aggregation. (UML notation depicts strong aggregation with a black diamond.) In either case, the Association is navigable to the stereotyped UML Class representing the user-defined type.

Example

For example, the IDL definition:

```
interface Vehicle {
    readonly attribute ManufacturerIfce manufacturer;
    readonly attribute short tireCount;
    attribute PropertyName vehicleId;
};
```

is represented in UML as in Figure 3-35.

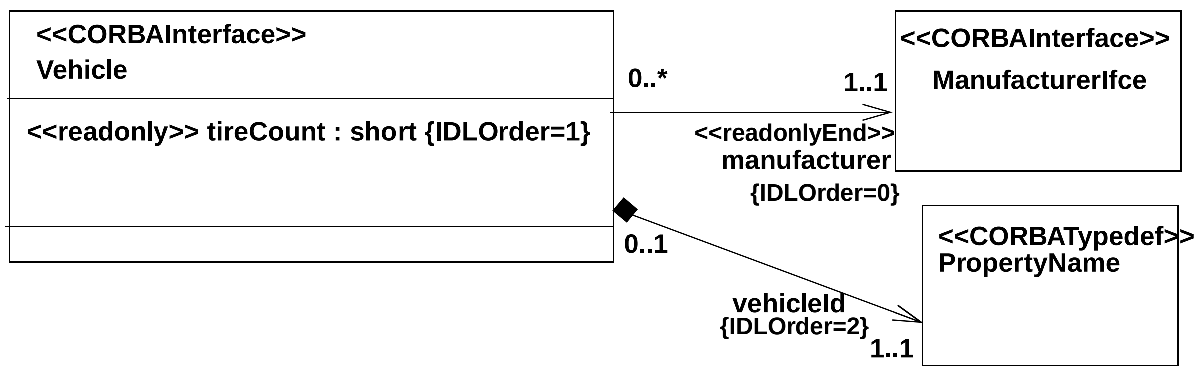

Figure 3-35 Interface Attribute Example

Note – The modeler may constrain multiplicity and choose aggregation kinds.

Complete Example 4

Contents

This chapter contains the following topics.

4.1 Introduction

This section is the result of a reverse engineering of the IDL of the OMG Task and Session Service, version 2. The IDL used as a basis is contained in OMG document dtc/99-08-05, and is included at the end of this chapter.

4.2 Approach

The reverse engineering was accomplished by using the Rational RoseCORBA reverse engineering tool, and then modifying the results to comply with this specification. RoseCORBA and the UML Profile for CORBA concern themselves only with class diagrams and the metamodel underlying class diagrams, and thus this example consists exclusively of class diagrams.

When graphically modeling a system via UML, it is generally advisable to avoid clutter in the diagrams. This approach results in more diagrams with fewer model elements per diagram.

4.3 Class Diagrams

4.3.1 The Class Diagrams for the OMG Task & Session Service

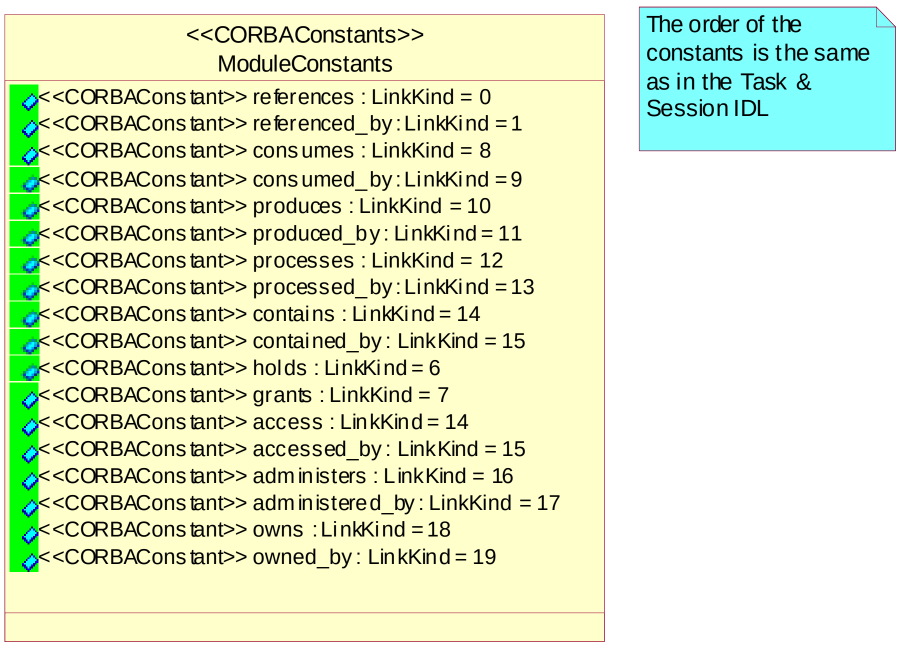

Figure 4-1 Constants

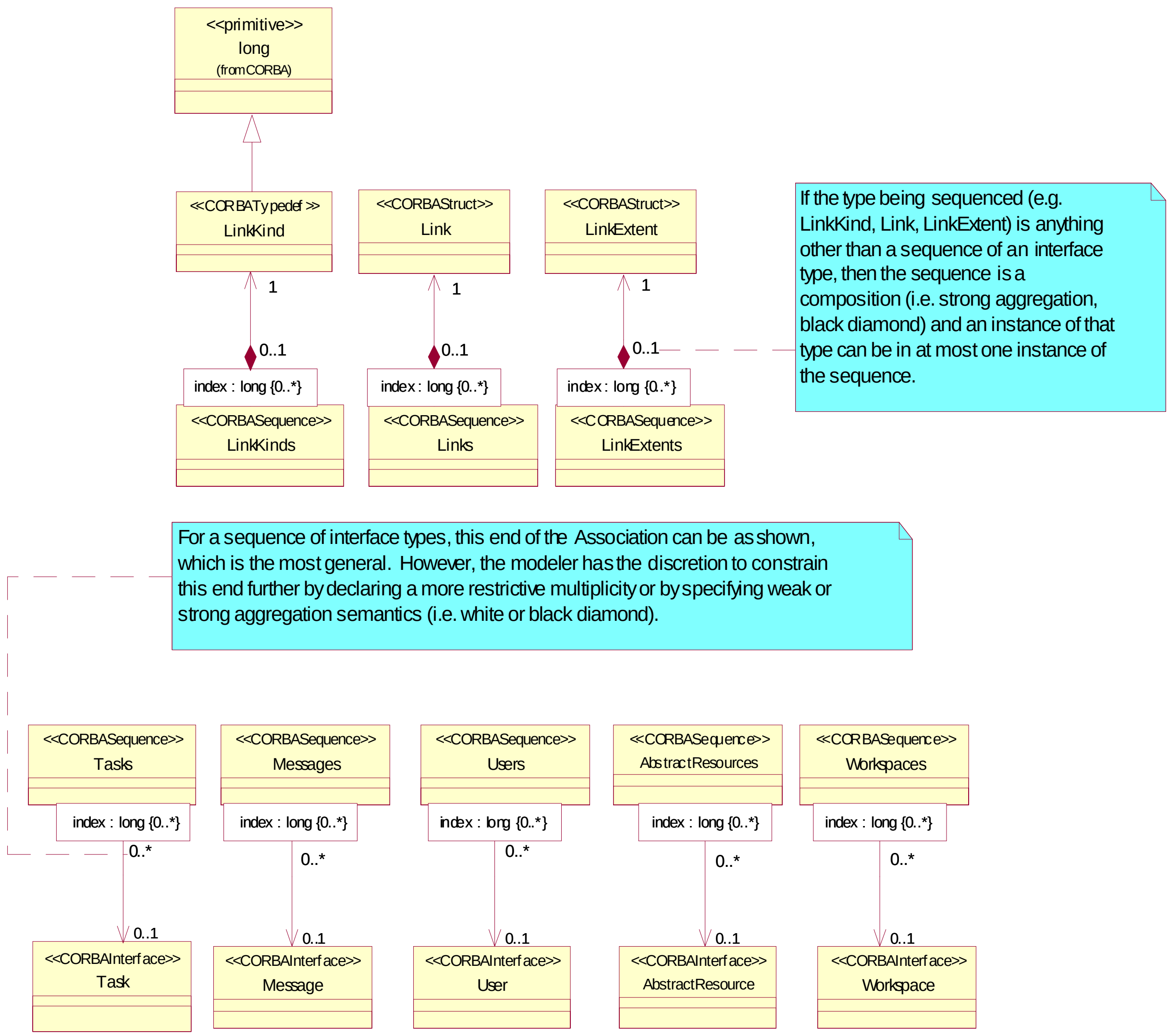

Figure 4-2 Typedefs and Sequences

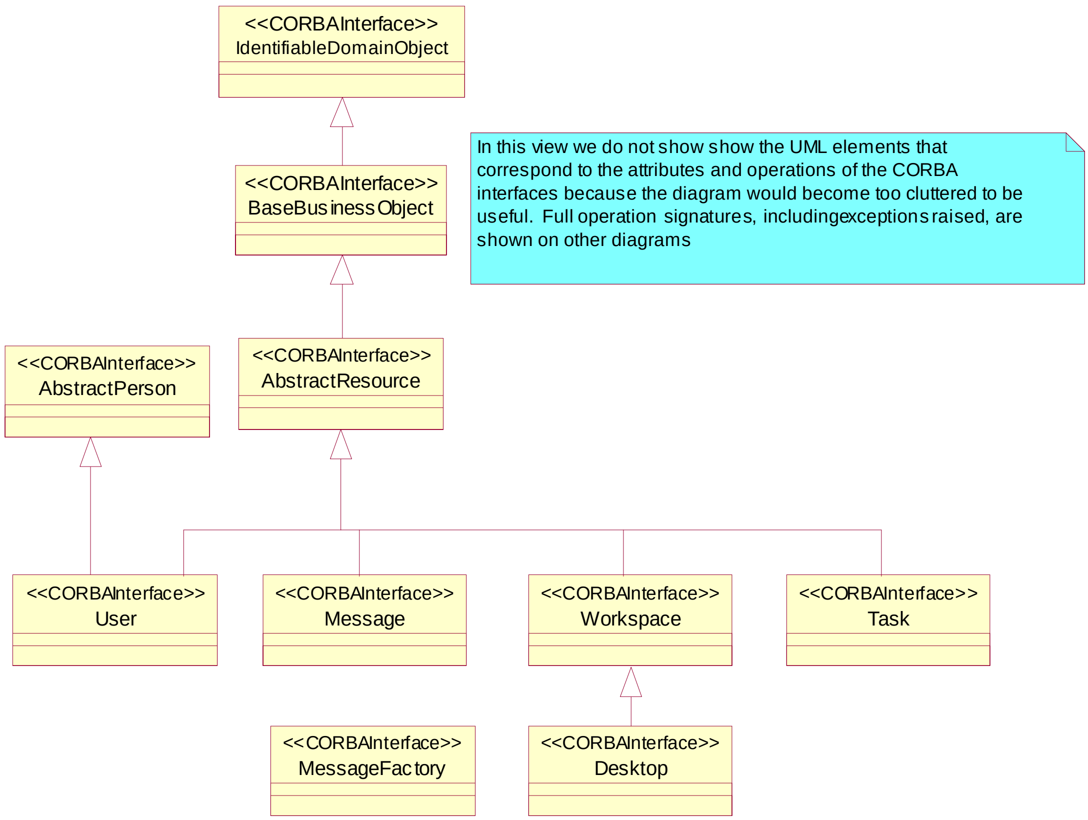

Figure 4-3 Interfaces--Inheritance View

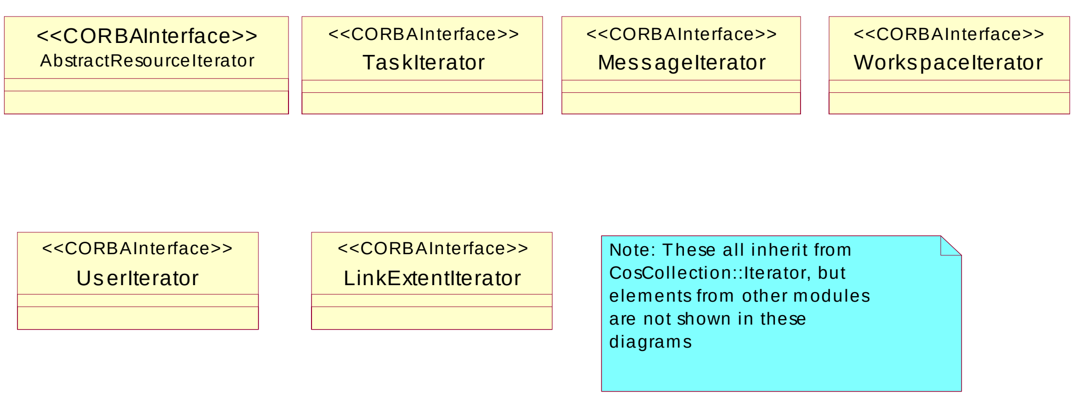

Figure 4-4 Interfaces--Iterators

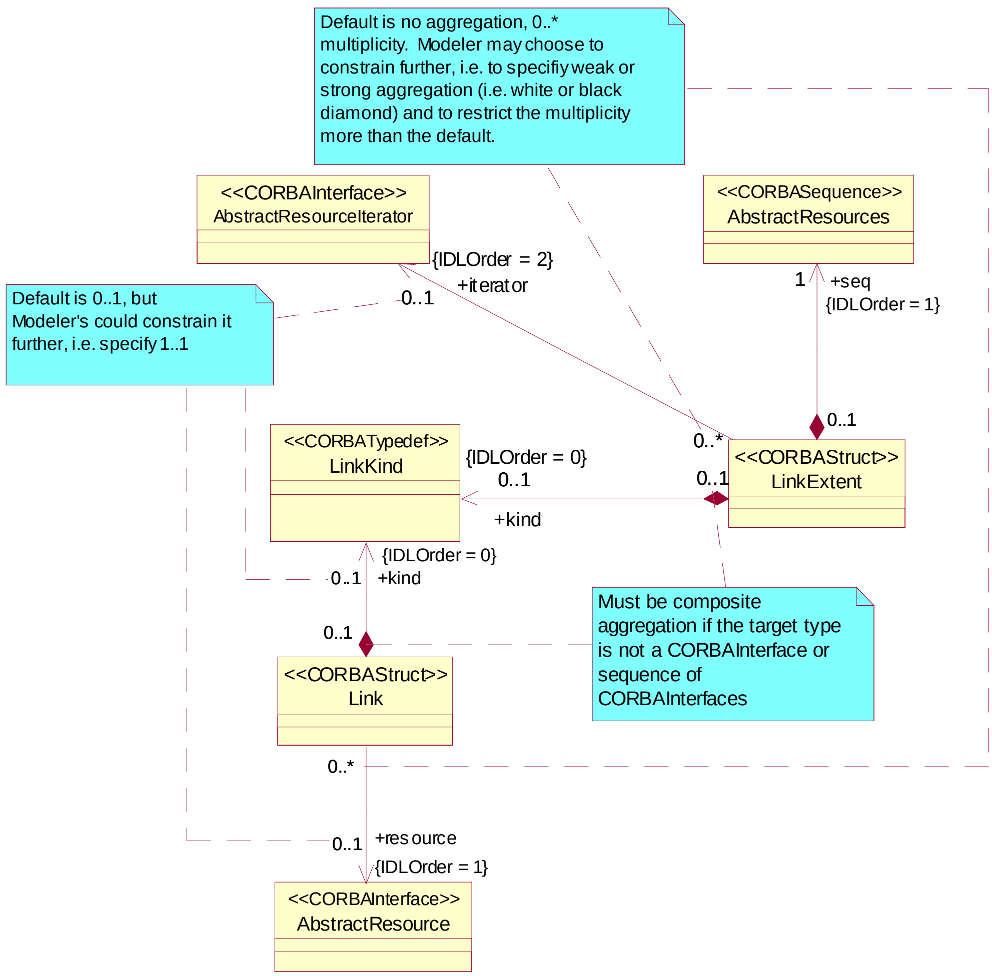

Figure 4-5 Structs

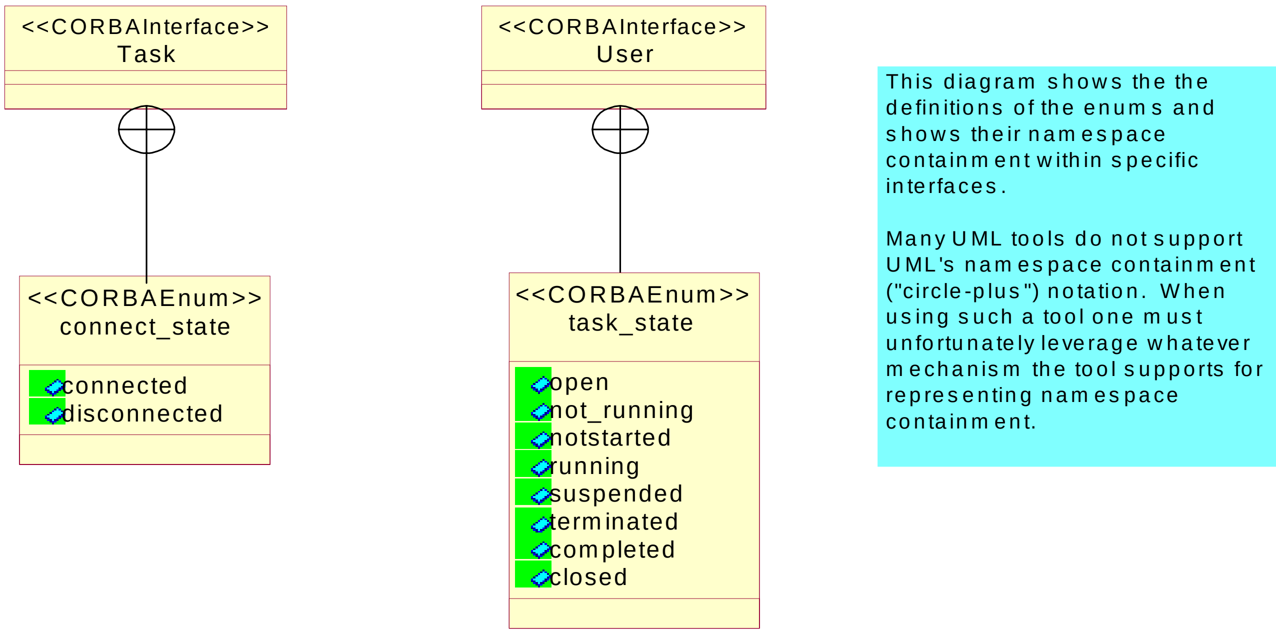

Figure 4-6 Enums

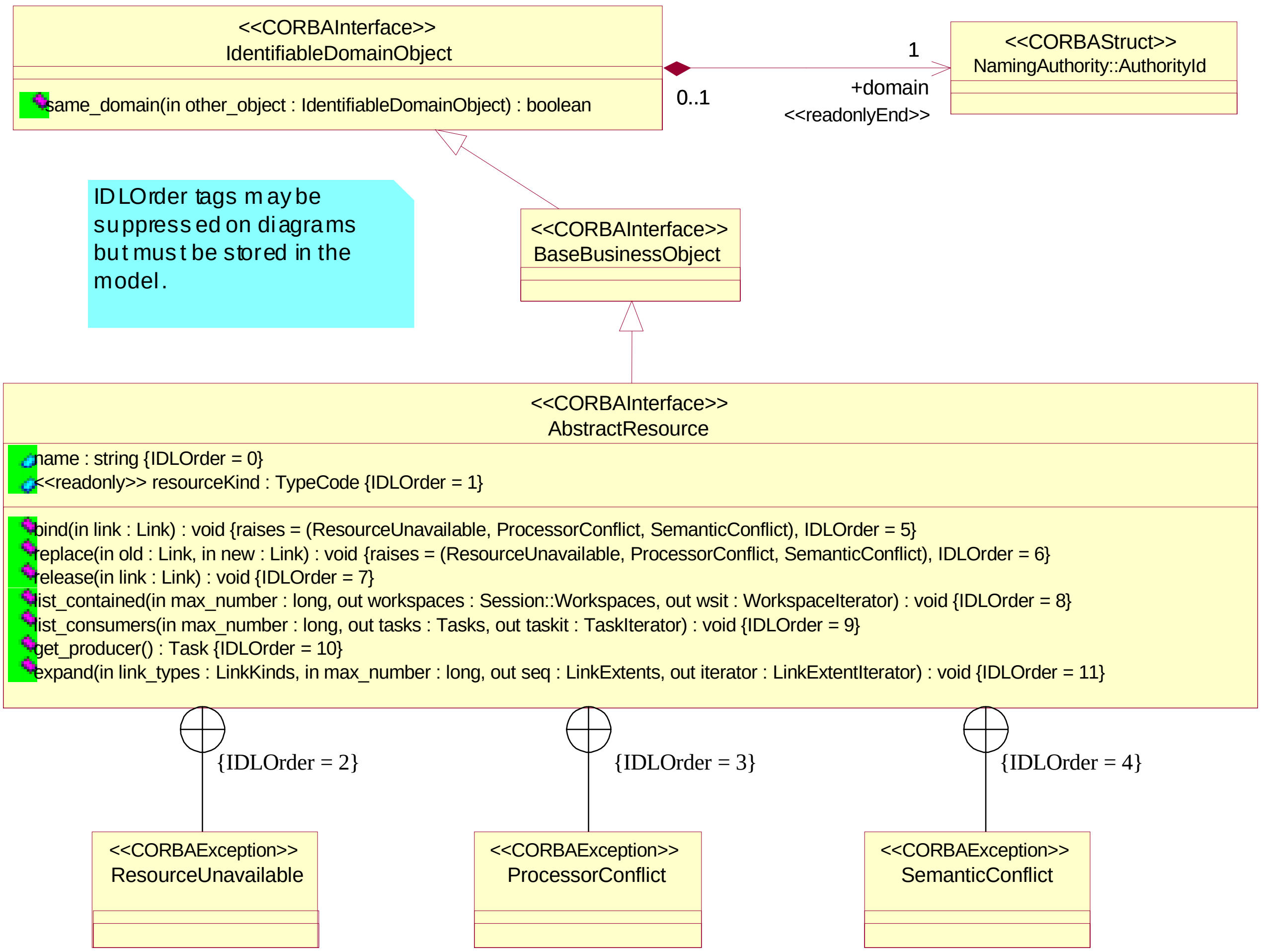

Figure 4-7 Interfaces--Full Signatures, Part 1, IDLOrder Tags Included

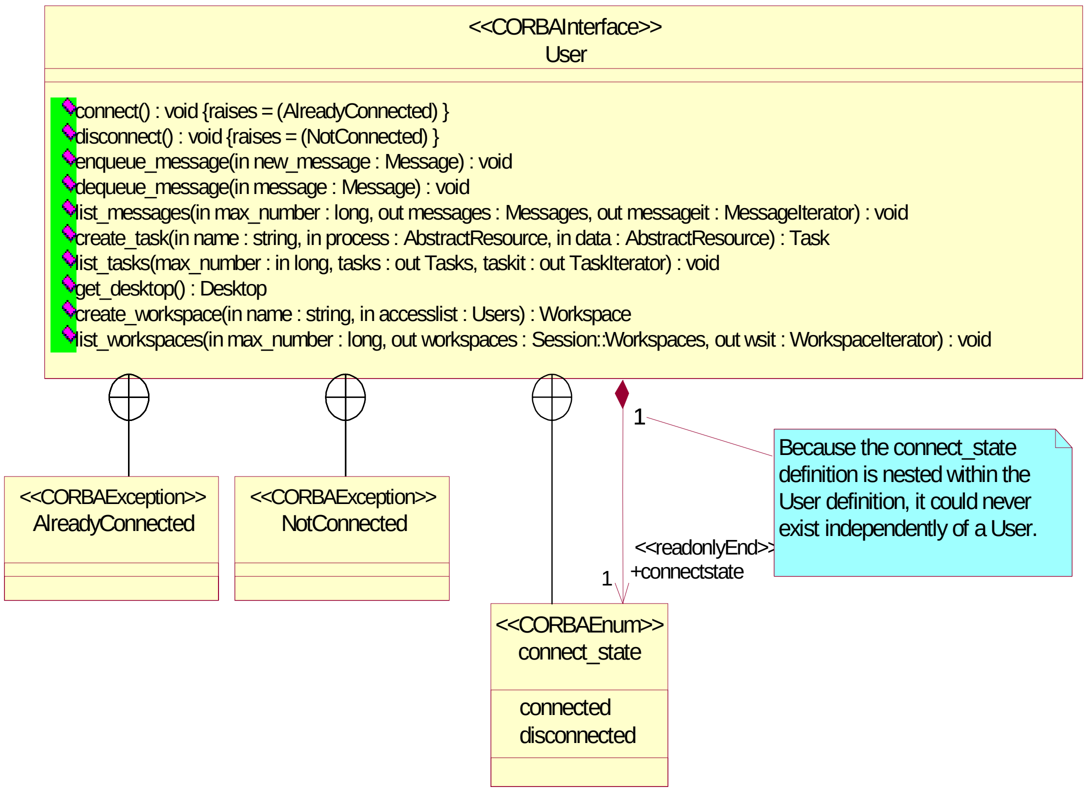

Figure 4-8 Interfaces--Full Signatures, Part 2, IDLOrder Tags Suppressed

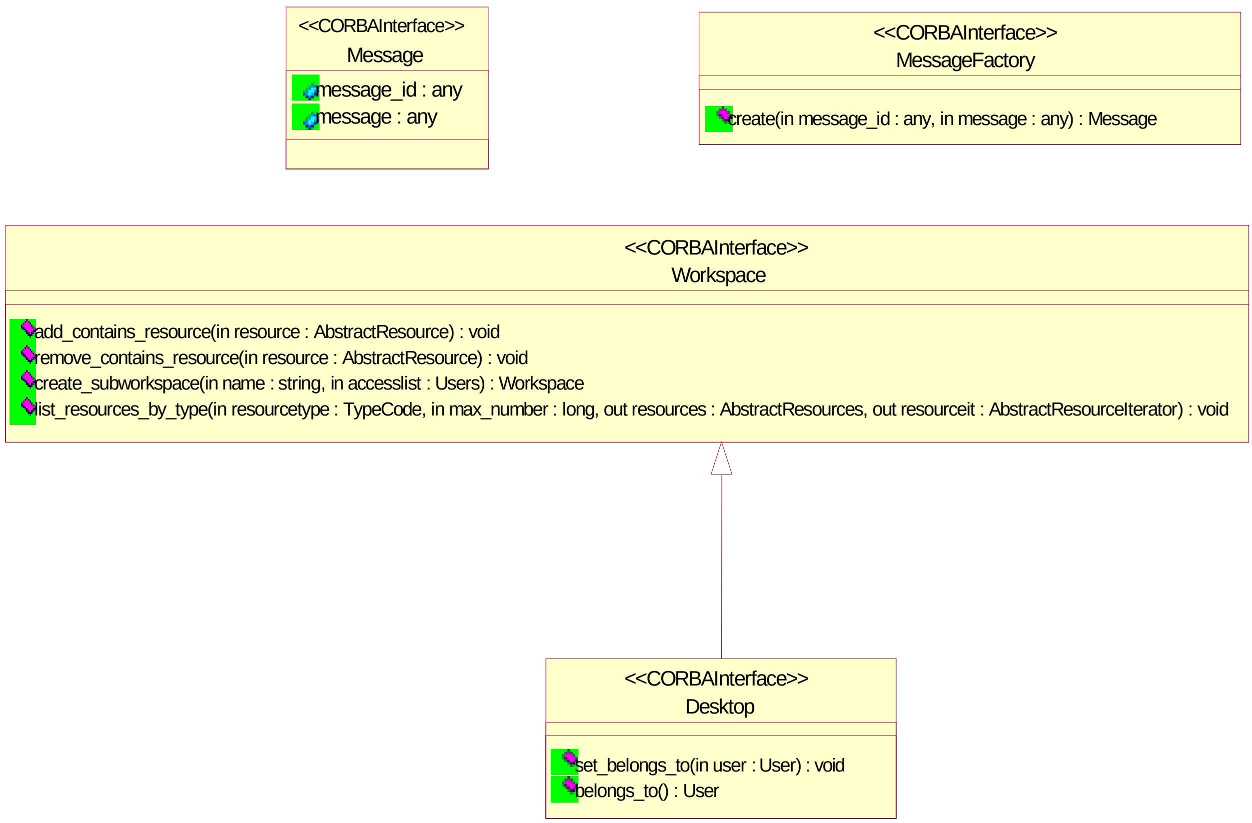

Figure 4-9 Interfaces--Full Signatures, Part 3

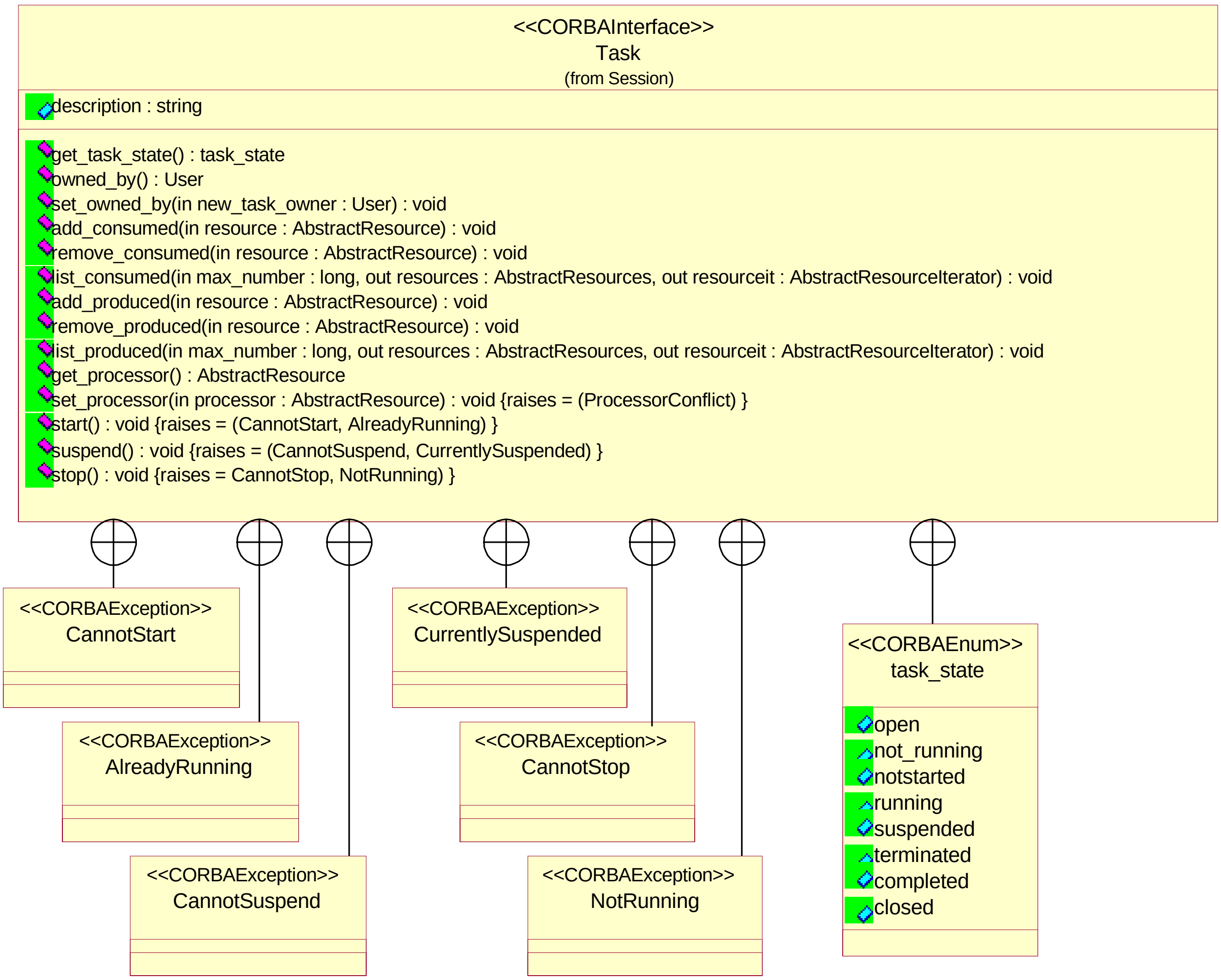

Figure 4-10 Interfaces--Full Signatures, Part 4

4.4 Task & Session IDL

4.4.1 The OMG Task & Session IDL (dtc/99-08-05)

```
// Task and Session RTF V2.0 Session.idl

#ifndef _SESSION_
#define _SESSION_

#include <CosLifeCycle.idl>
#include <CosObjectIdentity.idl>
#include <CosCollection.idl>
#include <NamingAuthority.idl>
#include <CosNotifyComm.idl>
#include <CosPropertyService.idl>

#pragma prefix "omg.org"
#pragma javaPackage "org.omg"

module Session {

    interface AbstractResource;
    interface Task;
    interface Workspace;
    interface AbstractPerson;
    interface User;
    interface Message;
    interface Desktop;

    typedef long LinkKind;

    // sequence definitions

    typedef sequence<Session::AbstractResource>AbstractResources;
    typedef sequence<Session::Task>Tasks;
    typedef sequence<Session::Message>Messages;
    typedef sequence<Session::User>Users;
    typedef sequence<Session::Workspace>Workspaces;
    typedef sequence<LinkKind>LinkKinds;

    // reference (abstract)
    const LinkKind references = 0;
    const LinkKind referenced_by = 1;

    // usage (abstract)
    const LinkKind uses = 2;
    const LinkKind used_by = 3;
```

```
// consumption
const LinkKind consumes = 8;
const LinkKind consumed_by = 9;

// production
const LinkKind produces = 10;
const LinkKind produced_by = 11;

// process
const LinkKind processes = 12;
const LinkKind processed_by = 13;

// containment
const LinkKind contains = 4;
const LinkKind contained_by = 5;

// rights (abstract)
const LinkKind holds = 6;
const LinkKind grants = 7;

// access rights
const LinkKind accesses = 14;
const LinkKind accessed_by = 15;

// administration rights
const LinkKind administers = 16;
const LinkKind administered_by = 17;

// ownership rights
const LinkKind owns = 18;
const LinkKind owned_by = 19;

struct Link {
    LinkKind kind;
    AbstractResource resource;
};

struct LinkExtent {
    LinkKind kind;
    AbstractResources seq;
    AbstractResourceIterator iterator;
};

typedef sequence<Session::Link>Links;
typedef sequence<LinkExtent>LinkExtents;

interface AbstractResourceIterator : CosCollection :: Iterator { };
interface TaskIterator : CosCollection :: Iterator { };
interface MessageIterator : CosCollection :: Iterator { };
interface WorkspaceIterator : CosCollection :: Iterator { };
```

```
interface UserIterator : CosCollection :: Iterator { };
interface LinkExtentIterator : CosCollection :: Iterator { };

interface IdentifiableDomainObject :
    CosObjectIdentity::IdentifiableObject
    {
    readonly attribute NamingAuthority::AuthorityId domain;
    boolean same_domain(
        in IdentifiableDomainObject other_object
    );
};

interface BaseBusinessObject :
    Session::IdentifiableDomainObject,
    CosLifeCycle::LifeCycleObject,
    CosNotifyComm::StructuredPushSupplier,
    CosNotifyComm::StructuredPushConsumer
    {
};

interface AbstractResource :
    BaseBusinessObject {
    attribute string name;
    readonly attribute TypeCode resourceKind;
    exception ResourceUnavailable{ };
    exception ProcessorConflict{ };
    exception SemanticConflict{ };
    void bind(
        in Link link
    ) raises (
        ResourceUnavailable,
        ProcessorConflict,
        SemanticConflict
    );
    void replace(
        in Link old,
        in Link new
    ) raises (
        ResourceUnavailable,
        ProcessorConflict,
        SemanticConflict
    );
    void release(
        in Link link
    );
    void list_contained (
        in long max_number,
        out Session::Workspaces workspaces,
        out WorkspaceIterator wsit
    );
    void list_consumers (
```

```
        in long max_number,
        out Tasks tasks,
        out TaskIterator taskit
    );
    Task get_producer(
    );
    void expand (
        in LinkKinds link_types,
        in long max_number,
        out LinkExtents seq,
        out LinkExtentIterator iterator
    );
};

interface AbstractPerson :
    CosPropertyService::PropertySetDef
    {
};

interface User  :
    AbstractResource,
    AbstractPerson,
    CosLifeCycle::FactoryFinder
    {
    enum connect_state {connected, disconnected};
    readonly attribute connect_state connectstate;
    exception AlreadyConnected {};
    exception NotConnected {};
    void connect()
        raises (AlreadyConnected);
    void disconnect()
        raises (NotConnected);
    void enqueue_message (
        in Message new_message);
    void dequeue_message (
        in Message message);
    void list_messages(
        in long max_number,
        out Messages messages,
        out MessageIterator messageit);
    Task create_task (
        in string name,
        in AbstractResource process,
        in AbstractResource data);
    void list_tasks (
        in long max_number,
        out Tasks tasks,
        out TaskIterator taskit
    );
    Desktop get_desktop ( );
    Workspace create_workspace (
```

```
        in string name,
        in Users accesslist
    );
    void list_workspaces (
        in long max_number,
        out Session::Workspaces workspaces,
        out WorkspaceIterator wsit
    );

};

interface Message : AbstractResource {
    attribute any message_id;
    attribute any message;
};

interface MessageFactory{
    Message create(
        in any message_id,
        in any message
    );
};

interface Workspace :
    AbstractResource,
    CosLifeCycle::FactoryFinder
    {
    void add_contains_resource(
        in AbstractResource resource
    );
    void remove_contains_resource(
        in AbstractResource resource
    );
    Workspace create_subworkspace (
        in string name,
        in Users accesslist
    );
    void list_resources_by_type(
        in TypeCode resourcetype,
        in long max_number,
        out AbstractResources resources,
        out AbstractResourceIterator resourceit
    );
};

interface Desktop:Workspace {
    void set_belongs_to(
        in User user
    );
    User belongs_to();
};
```

```
        interface Task : AbstractResource {
            exception CannotStart {};
            exception AlreadyRunning {};
            exception CannotSuspend {};
            exception CurrentlySuspended {};
            exception CannotStop {};
            exception NotRunning {};
            attribute string description;
            enum task_state {
                open, not_running, notstarted, running,
                suspended, terminated, completed, closed
            };
            task_state get_task_state();
            User owned_by();
            void set_owned_by (
                in User new_task_owner
            );
            void add_consumed(
                in AbstractResource resource
            );
            void remove_consumed(
                in AbstractResource resource
            );
            void list_consumed (
                in long max_number,
                out AbstractResources resources,
                out AbstractResourceIterator resourceit
            );
            void add_produced(
                in AbstractResource resource);
            void remove_produced(
                in AbstractResource resource);
            void list_produced (
                in long max_number,
                out AbstractResources resources,
                out AbstractResourceIterator resourceit
            );
            AbstractResource get_processor( );
            void set_processor(
                in AbstractResource processor
            ) raises (
                ProcessorConflict
            );
            void start ( ) raises (CannotStart, AlreadyRunning);
            void suspend ( ) raises (CannotSuspend, CurrentlySuspended);
            void stop ( ) raises (CannotStop, NotRunning);
        };
    };

    #endif /* _SESSION_ */
```

Conformance Issues A

This appendix specifies the conformance points for this specification.

A.1 Conformance

There are two kinds of conformance that can be defined with respect to a UML profile. One kind is conformance by UML modeling tools and the other kind is conformance by specific UML-based object models.

A.1.1 Versions

All conformance points below require implementations to model CORBA IDL as specified in CORBA 2.4, using UML models as specified in UML 1.3.

A.1.2 Conformance by Specific UML Object Models

A specific UML object model either conforms with the defined UML Profile for CORBA or it does not. There are no categories of this kind of conformance. A UML object model conforms with the profile if it restricts itself to the identified subset of the UML metamodel and satisfies all constraints imposed by the profile.

A.1.3 Conformance by UML Modeling Tools

The term *all constructs defined by the profile* is used several times in the following Conformance points. This term is defined to mean all UML constructs that are part of the profile's identified subset of UML plus all extensions to that subset that the profile defines. This term thus includes UML constructs that are part of the identified subset but that the profile does not extend. For example, the profile does not extend the UML Core construct Constraint but, because Constraint is part of the identified subset, the term includes Constraint.

A UML modeling tool is considered to be a Conformant *simple modeling tool* for the UML Profile for CORBA if it supports expression of all constructs defined by the profile. The tool must support such expression via the notation specified by UML 1.3 as applied by the UML Profile for CORBA notation guidelines.

A UML modeling tool is considered to be a Conformant *forward-engineering tool* for the UML Profile for CORBA if it can perform a transformation in which the source is any arbitrary object model expressed in terms of the profile and the target is CORBA IDL, where the transformation satisfies the definition of the profile. The parser must be able to detect violations of the profile's constraints and produce error messages that explain the violations. The tool must permit the source object model to use all constructs defined by the profile.

A UML modeling tool is considered to be a Conformant *reverse-engineering tool* for the UML Profile for CORBA if it can perform a transformation in which the source is an object model expressed in either CORBA IDL or contained in a CORBA interface repository and the target is an object model expressed in terms of the profile, where the transformation satisfies the definition of the profile. If the source is CORBA IDL, the reverse-engineer tool must be able to detect and produce error messages about errors in IDL syntax. The tool must support all legal IDL syntax.

Index

Index