CORBA Components

Joint Revised Submission

BEA Systems, Inc.
Cooperative Research Centre for Distributed Systems Technology
Expersoft Corporation
Genesis Development Corporation
IBM Corporation
Inprise Corporation
IONA Technologies, PLC
Oracle Corporation
Rogue Wave Software, Inc.
Unisys Corporation

Supported by:
Fujitsu, Ltd.
Hewlett-Packard Corporation
Sun Microsystems, Inc.

OMG TC Document orbos/99-02-05
March 1, 1999

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction 1

The following companies are pleased to jointly submit this specification in response to the CORBA Component Model RFP (Document orbos/97-06-12):

- BEA Systems, Inc.
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Expersoft Corporation
- Genesis Development Corporation
- IBM Corporation
- Inprise Corporation
- IONA Technologies, PLC
- Oracle Corporation
- Rogue Wave Software, Inc.
- Unisys Corporation

Recognizing the importance of aligning this specification with Enterprise Java Beans, the submitters are pleased to acknowledge the cooperation of:

- Sun Microsystems

In addition, we also acknowledge support from:

- Fujitsu, Ltd.
- Hewlett-Packard, Inc.

1.1 Overview

The submitters believe that a CORBA component model should focus on the strength of CORBA as a server-side object model. To that end we have chosen to concentrate on those issues that must be addressed to provide a server facility rather than a client facility. We compare this model to the Enterprise Java Bean specification which was released by JavaSoft after the OMG's Component RFP was issued rather than the Java Beans model requested by the original RFP.

The submitters believe that the Java Beans model is inappropriate for server side development.

Just as Sun chose to define a different component model with Enterprise Java Beans (EJB) than its Java Bean predecessor, we chose to define CORBA components as a server-side model which more closely aligns with EJB than Java Beans. The component model defined by this specification has the following characteristics:

- It defines extensions to IDL to support the definition of CORBA components and the relationships between them.
- It introduces CIDL, a language similar to IDL, as a mechanism for defining servant implementations that enhances the ability to do automatic code generation on behalf of the developer.
- It defines extensions to the CORBA core object model to introduce the concept of components to the OMA.
- It defines interfaces necessary to support navigation among the multiple interfaces supported by a CORBA component.
- It defines a mechanism for tailoring CORBA components prior to deployment using both metadata defined by the component model and runtime properties which can be tailored using a design tool.
- It introduces a deployment model to CORBA using XML to describe the run time properties of a CORBA component.
- It defines a container model for introducing system services into the runtime environment of a CORBA component.
- It defines locality constrained interfaces for a component to interact with its container.
- It introduces the container programming model, a higher level abstraction of the POA and the CORBA services for use by the developer and defines the container as a simplified set of policies derived from the Portable Object Adaptor (POA).
- It defines interfaces to manage object activation and passivation derived from the POA policies selected.
- It defines policies which support a simplified version of CORBA transactions. These policies provide transaction control independent of the component implementation and integrate synchronization between object state and persistent storage prior to commit processing. They also permit the component itself to control transaction demarcation.

- It defines policies for managing servant lifetimes to optimize resource usage within a process thereby enhancing the scalability of a compliant implementation.
- It defines security policies which provide authorization based on role as described by the CORBA Security Service.
- It defines policies which provide persistent state management based on the POA for all CORBA components, either with application assistance or automatically in conjunction with the CORBA Persistent State Service.
- It defines a mapping to Enterprise Java Beans which makes it possible for an EJB to be supported as a CORBA component within a container which provides activation, transactions, security, events, and persistence.

1.2 Relationship to other CORBA Technology

CORBA components extend the CORBA core object model and introduce a deployment model into the OMA. They also provide a higher level of abstraction of CORBA and object services, greatly simplifying CORBA programming.

1.2.1 CORBA Core and Object Services

CORBA Components extend the core object model through the introduction of **component** types and support for multiple interfaces. Components use services above the core, specifically the POA, transactions, security, events, and persistence in a specialized way to offer the programmer a simpler programming abstraction. The submitters believe that this abstraction is suitable for a broad spectrum of CORBA applications.

1.2.2 Business Objects Interoperability Initiative

The Business Objects Interoperability Initiative seeks a framework suitable for deploying a new category of CORBA objects, designated as business objects. It does so by defining a meta-model which introduces the notion of business semantics to the behavior description of these CORBA objects. The initiative also seeks a technology mapping of these concepts to the CORBA model, including the CORBA services.

CORBA components can serve as an alternative technology mapping of this business objects architecture, since it incorporates many of the design patterns used by business objects in support of the various CORBA services. CORBA components, however, are not the same as business objects because they do not of themselves define any of the business semantics desired for the business object model.

1.2.3 UML and the Meta Object Facility

This specification of CORBA components defines a meta-model based on UML and a mapping of that meta-model to the MOF. The meta-model includes the component extensions to IDL and the Interface Repository as well as the component deployment model defined by this specification. This meta-model requires no changes to UML.

1.3 Guide to the Submission

The submission is organized as follows:

- Chapter 2 provides a mapping of the submission to the requirements specified in the CORBA Components RFP (orbos/97-06-12).
- Chapter 3 contains an overview of the architecture for CORBA components which introduces the major concepts that are further described in the ensuing chapters.
- Chapter 4 introduces core changes to support locality-constrained interfaces which are necessary to define the CORBA component model.
- Chapter 5 provides a description of the abstract model for Components including the changes to IDL and the CORBA core.
- Chapter 6 describes the component implementation framework which supports the component model and integrates the use of PSS.
- Chapter 7 defines the programmer's view of the container model with emphasis on the contract between the container and the server programmer.
- Chapter 8 specifies the architecture of the container with emphasis on the contract between the container provider and the ORB, POA, and the CORBA services.
- Chapter 9 provides a description of the deployment model, including packaging and distribution.
- Chapter 10 provides a description of the component meta model and its realization in UML and the MOF.
- Chapter 11 provides a mapping of the CORBA component model to Enterprise Java Beans (EJB).
- Chapter 12 defines the changes to the C++ language mappings required by CORBA components.
- Chapter 13 defines the changes to the Java language mappings required by CORBA components.
- Chapter 14 provides instructions to the editor of the specific changes to CORBA and the CORBA services introduced by this specification.
- Chapter 15 provides a description of the compliance criteria for conforming implementations.

In addition to the normative parts of the specification, several appendices are provided as clarifications:

- Appendix A summarizes the IDL introduced by this specification. All of this IDL has been introduced in the normative portion of the specification.
- Appendix B summarizes the XML DTDs introduced by this specification. All of this XML has been introduced in the normative portion of the specification.

- Appendix C contains the IDL for the MOF metamodels of the Interface Repository, including the component extensions, and the component packaging and deployment metamodel as well as the XML generated using the XMI standard for metadata interchange.
- Appendix D compares CORBA components to other component models including Java Beans and Rapide.
- Appendix E contains references to other work in this area.

1.4 Proof of Concept

The specification presented here is based on the extensive experience the submitting companies have had over the past year with their “experimental” and/or commercial implementations, e.g. BEA’s M3 product implements many of the interfaces defined for the CORBA container albeit with different API syntax. Many of the alternative designs that were considered have actually been implemented and tried by many users. The final choices that are embodied in this submission were made based upon user and vendor experience.

Shipping product which implements this specification can be expected to be made available almost concurrently with its final approval.

1.5 Conventions

IDL appears using this font.

```
XML appears using this font.
```

Language Mapped code appears using this font.

Important Reminders appear using this font.

In some chapters, rationale appears using this font.

In various places a few issues are highlighted. These are mostly areas where we have discovered that some additional clarification may be needed.

Please note that any change bars have no semantic meaning. They show the places that final edits were applied to the last reviewed draft submission. They are present for the convenience of the submitters (and the editor who didn’t want to have to re-edit the entire document to remove change bars and maintain two synchronized copies) so that the final edits can be identified.

1.6 Submission Contact Points

All questions about this submission should be directed to:

Ed Cobb (Editor)
BEA Systems Inc.
2315 North 1st St.
San Jose, CA 95131
USA
phone: +1 408 570 8264
fax: +1 408 570 8910
email: ed.cobb@beasys.com

Keith Duddy
CRC for Distributed Systems Technology
University of Queensland
Brisbane 4072, Queensland
Australia
phone: +61 7 3365 4310
fax: +61 7 3365 4311
email: dud@dstc.edu.au

Shahzad Aslam-Mir
Expersoft Corporation
5825 Oberlin Drive
San Diego, CA 92121
phone: +1 619 824 4128
fax: +1 619 824 4110
email: sam@expersoft.com

David Frankel
Genesis Development Corporation
741 Santiago Court
Chico, CA 95973
USA
phone: +1 530 893 1100
fax: +1 530 893 1153
email: dfrankel@gendev.com

Jim Rhyne
IBM Canada Ltd. 2G/846/1150/TOR
1150 Eglington Ave. E.
Toronto, Ontario M3C 1H7
Canada
phone: +1 416 448 4383
fax: + 1 416 448 4414
email: jrhyne@us.ibm.com

David Curtis
Inprise Corporation
951 Mariner's Island Blvd.
San Mateo, CA 94404
USA
phone: +1 650 358 2447
fax: +1 650 286 2475
***email*:** dcurtis@inprise.com

Jeff Mischkinsky
Inprise Corporation
951 Mariner's Island Blvd.
San Mateo, CA 94404
USA
phone: +1 650 358 3049
fax: +1 650 286 2475
***email*:** jeffm@inprise.com

Martin Chapman
IONA Technologies, PLC
The IONA Building
Shelbourne Rd.
Dublin 4,
Ireland
phone: +353 1 637 2000
fax: +353 1 637 2888
email: mchapman@iona.com

Garrett Conaty
IONA Technologies, PLC
The IONA Building
Shelbourne Rd.
Dublin 4,
Ireland
phone: +353 1 637 2000
fax: +353 1 637 2888
email: gconaty@iona.com

Glenn Seidman
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
phone: +1 650 506 5823
fax: +1 650 654 6208
email: gseidman@us.oracle.com

Jim Trezzo
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
USA
phone: +1 650 506 8240
fax: +1 650 654 6208
email: jtrezzo@us.oracle.com

Patrick Thompson
Rogue Wave Software
815 NW 9th St.
Corvallis, OR 97330
USA
phone: +1 541 754 3189
fax: +1 541 758 4761
email: thompson@roguewave.com

Sridhar Iyengar
Unisys Corporation
25725 Jeronimo Road
Mission Viejo, CA, 92691
USA
phone: +1 714 380 5692
fax: +1 714 380 6600
email: sridhar.iyengar2@unisys.com

Mapping to RFP Requirements 2

2.1 Mandatory Requirements

- ***Responses shall specify a component model for CORBA systems. This model shall be structured as a natural extension of the existing CORBA object model, and shall be informed by experiences with other successful component models, such as JavaBeans and COM.***

  The component model specified in this submission is based on extensions to the CORBA model. The submitters have focussed on a server side model and have considered input from Enterprise Java Beans, COM+, and existing CORBA-based products.

- ***Responses shall define the elements of a component model, and concrete expressions of these elements in terms of CORBA technology.***

  All elements of the abstract model are specified using IDL with extensions to support the component architecture. Packaging and deployment have not been previously considered in CORBA specifications and, based on similar work in the W3C, are specified using XML. The container specification is based on the Portable Object Adaptor (POA) and uses a new **local** IDL construct to define locality-constrained interfaces.

- ***Responses shall build upon existing specifications, and be aligned with other simultaneously emerging specifications.***

  The specification is based on CORBA 2.3 and the current levels of CORBA transactions (1.1), CORBA security (1.2), and CORBA notification (1.1). It integrates work in process for CORBA persistence.

2.1.1 Component Model Elements

- ***Responses shall clearly define the concept of component type and the structure for a component typing system, and shall specify mechanisms for establishing and expressing component type identity.***

These mechanisms are defined as part of the abstract model in Chapter 5. The specification address both the type system for components and the notion of component identity.

- ***Responses shall define a concrete concept of component instance identity, and a reliable means for determining whether two interface references belong to the same component instance.***

  These mechanisms are defined as part of the abstract model in Chapter 5. Operations are defined on a new CORBA metatype, **CORBA::Component,** which provide the necessary mechanisms for determining if two interfaces are within the same component.

- ***Responses shall describe the life cycle of a component, and specify interfaces and mechanisms for managing its life cycle.***

  Component life cycle is described as part of the abstract model in Chapter 5. Life cycle operations are also defined as part of the container APIs in Chapter 7.

- ***Responses shall describe the association between a component and its interfaces, and their relative life cycles. These descriptions shall be consistent with responses to the Multiple Interfaces RFP.***

  These descriptions are provided as part of the abstract model in Chapter 5. Since the ORBOS Task Force voted to terminate the Multiple Interfaces RFP in January 1999, consistency with that specification is no longer applicable.

- ***Responses shall specify interfaces for exposing and managing component properties. Properties are an externally accessible view of a component's abstract state that can be used for design-time customizing of the component, and which support mechanisms for notification (event generation) and validation when a property's value changes. Responses shall define the relationship between component properties and IDL interface attributes, if any.***

  These descriptions, which are based on an extended version of CORBA attributes, are provided as part of the abstract model in Chapter 5. Although a mechanism for distinguishing design time from run time is not mandated by this specification, such a mechanism can be implemented using the configuration architecture defined in Chapter 5. Because the component model is designed for the server, the property-change notification system inherent in client component models like Java Beans was **not** adopted. Instead a more robust event mechanism based on CORBA notification is specified.

- ***Responses shall specify interfaces and mechanisms for serializing a component's state and for constructing a component from serialized state. The serialization mechanism shall be suitable for storage and retrieval, and for externalizing state over communication channels. To the extent possible, this serialization mechanism shall be aligned with other existing or emerging serialization mechanisms, such as the externalization service, proposed streaming mechanisms for passing objects by value, proposed mechanisms in Messaging Service responses, and so on. The intent of this requirement is to avoid further redundant serialization interfaces in CORBA specifications.***

Serialization, as it exists in JavaBeans, is not applicable to CORBA components. It is used in Enterprise Java Beans for deployment descriptors, but it is Sun's stated intention to move to XML for that purpose. CORBA components use XML for their deployment descriptors as defined in Chapter 9. The mechanism for serializing a component's abstract state is based on the techniques proposed by the Persistent State Service submissions which use state declarations as part of the abstract model to provide representations of that state.

- ***Responses shall specify interfaces and mechanisms for generating events, and for installing arbitrary event handlers (listeners) for specific events generated by components. The event mechanism shall be coordinated with the property mechanisms to support event generation when property values are modified. The relationship between this component model's event mechanism and the existing CORBA Event Service shall be clearly defined. If a response does not make use of the existing Event Service, it shall provide rationale for this decision.***

  The event mechanism is defined as part of the abstract model in Chapter 5 and permits arbitrary event handlers to consume events generated by the component. Its architecture is based on the CORBA notification service which is derived from the CORBA event service. This provides a robust event distribution mechanism more scalable and functional than the event mechanism provided by other component models such as JavaBeans and COM.

2.1.2 Requirements for Component Description Facility

- ***Responses shall specify an information model that describes components. In conjunction with the information model, responses shall specify a set of interfaces for a programmatic representation of this information model and a textual representation (i.e. a description language) for the information model. This language may be an extension to IDL or a complementary adjunct to IDL. Responses shall provide rationale for their decision regarding the form of the language and its relationship to IDL. The information model shall capture all the salient features of components.***

  The component information model is addressed by this specification in multiple ways:

  - IDL extensions are defined in Chapter 5 to capture the designer's intent and to allow component tools to perform code generation.
  - A Component Implementation Definition Language (CIDL) is introduced in Chapter 6 to define abstract state for container-managed persistence and to define other properties of the component's implementation in the server.
  - Run-time descriptions necessary to create instances of components and their deployment characteristics are defined in Chapter 9 and described using XML based on similar work being done in the W3C.
  - A MOF-based meta-model is provided for the abstract component model in Chapter 10.

- ***Responses shall specify how component descriptions are stored in a repository. The relationship between this repository and existing CORBA repositories, including the Interface Repository, Implementation Repository,***

and the Meta-data repository shall be clearly defined. The information models supported by the description language and the repository shall be completely isomorphic. The mapping between the description language and the repository contents shall be reflexive.

This specification provides extensions to the Interface Repository (IR) which contain the additional information associated with components. These extensions are defined as part of the abstract model in Chapter 5. The meta-model defined in Chapter 10 is based on the MOF.

2.1.3 Requirements for Programming Model

- ***Responses shall describe a mapping from the component description information to a concrete programming model, and define how that programming model is expressed in programming languages that support IDL mappings.***

  This specification defines the abstract model as extensions to IDL in Chapter 5. Where required, new language mappings are defined in Chapter 12.

- ***The mapping shall automate the generation of as many programming details as reasonably possible. For example, if the information in the component description contains a complete description of a component's state, the responses shall describe how methods for serializing that state will be generated from the description.***

  The specification was designed with the goal of automatic code generation. Techniques for creating factory code as well as automating persistence were introduced into the model. Based on the experience of the submitters, we believe such automation is feasible with the techniques defined in this specification. This is elaborated in Chapter 6.

- ***Responses shall specify interfaces and mechanisms so as to maximize the portability of component implementation code between compliant implementations of the specification. To this end, responses shall clearly define the relationship between elements of component model and the interfaces specified in the Enhanced ORB Portability specification, particularly the POA and its related interfaces. Responses shall specify how the behaviors and policies supported by the POA interfaces apply to components, and describe the relationships between servants and component implementations. If possible, responses shall define how implementations of objects required by the POA, such as servant managers, may be automatically generated from component descriptions.***

  The container architecture defined in Chapter 8 is derived by specializing the Portable Object Adaptor (POA). The POA policies used by the containers are clearly identified as are extended versions of POA interfaces which provide additional functionality.

- ***Responses shall specify how components can be passed as value parameters in CORBA requests. This specification shall be aligned with responses to the Objects by Value RFP.***

CORBA Components cannot be **valuetypes** so they cannot be passed as value parameters in CORBA requests. Where needed, a component developer may provide operations and attributes which produce **valuetypes** that encapsulate all or part of the component's state and behavior.

2.1.4 Requirements for Mapping to JavaBeans

- ***Responses shall specify a mapping from the proposed component model to the JavaBeans component model. Responses shall define and address the mapping between the intersection of the two component models (i.e. it is not a requirement that the two models be isomorphic).***

  The component model defined by this specification is a superset of the Enterprise Java Beans component model. It contains additional function beyond the EJB 1.0 specification including events and tighter integration with the CORBA object model. The mapping to EJB is described in Chapter 11.

- ***The mapping shall permit a CORBA component to present itself as a JavaBean to Java programs and application building tools based on JavaBeans.***

  The specification defines a mechanism for a CORBA component written in Java to comply with the Enterprise Java Beans level of function and describes the constraints on a Java CORBA component to be an EJB.

- ***The mapping shall support automatic generation of elements required to effect the mapping.***

  Deployment descriptors in EJB are currently specified as **.jar** files. It is Sun's intent to utilize XML for this purpose in the future. Component descriptors are specified using XML. Considerations for converting between component XML and EJB **.jar** files are covered in Chapter 11.

- ***The mapping shall support both run-time and design-time needs. Responses shall describe how component descriptions are mapped to BeanInfo structures, so that visual application building tools that rely on BeanInfo can be used to configure and assemble CORBA components and JavaBeans interchangeably.***

  The specification considers both design time and runtime. Since the component model maps to EJB rather then JavaBeans, mapping to BeanInfo structures are neither required nor provided. The configuration architecture defined in Chapter 5 provides mechanisms to distinguish between design time and run time. Where possible, the submitters have adopted EJB syntax to minimize impact on existing or planned EJB tools.

- ***The mapping shall maximize interoperability between features of the CORBA component model and the JavaBeans model.***

  All features of the EJB model are accommodated in the CORBA component model, either directly or by the EJB to CORBA components mapping in Chapter 11.

- ***The version of the Java Beans specification that shall be used is JavaBeans 1.0 Revision A unless it is superseded by a revised specification issued before the submission due date. The specification is available at <http://splash.javasoft.com/beans/beans.100A.pdf>.***

This specification is based on V1.0 of the Enterprise Java Beans specification **<ftp://ftp.javasoft.com/docs/ejb/ejb.10.pdf>**. In-process changes beyond the 1.0 level of the specification will have to be reconciled during the P-spec RTF process.

- ***The JavaBeans specification is still under development and significant new features are being considered. Submitters should track these developments. Information about new draft specifications is available on the JavaBeans web page at <http://splash.javasoft.com/beans>.***

  Since CORBA components is based on the Enterprise Java Beans specification, not the JavaBeans specification, this requirement is **not** applicable as written. However the submitters have attempted to track changes to the EJB 1.0 specification and anticipate them within this submission.

2.1.5 Security Requirements

- ***What, if any, are the security sensitive objects that are introduced by the proposal?***

  Distributed components systems introduce no new security requirements beyond those required of distributed object systems. All objects introduced in this proposal can have CORBA security policies applied to them in the same way that other objects participate.

- ***Which accesses to security-sensitive objects must be subject to security policy control?***

  The choice of objects subject to security policy control is up to the security administrator at each site. CORBA components place no constraints on the application of any security policy by any administrator.

- ***Does the proposed service of facility need to be security-aware?***

  Under normal operating conditions, security policy may be set on individual components and their interfaces by the administrator and it will be enforce by the component container using CORBA security. The container API framework defined in this specification (Chapter 7) allows the component implementation to perform additional security checking by testing security roles against the credentials in effect for CORBA security when an operation is dispatched.

- ***What CORBAsecurity level and options are required to protect an implementation of this proposal?***

  In general, this is up to the security administrator, however we recommend that security level 2 be used with authentication, and authorization. Auditing policy is at the discretion of the administrator as is message protection (except where export restrictions apply). Note that CORBA security provides no standard way to use SSL to establish client credentials.

- ***What default policies should be applied to security sensitive objects introduced by the proposal?***

  CORBA components introduces no new unique security requires beyond those of today's distributed object systems. Security administrators can choose the level of protection they desire for any and all of the objects defined by CORBA components.

- ***Of what security considerations must the implementors of your proposal be aware?***

  A design goal of CORBA components is compatibility and interoperability with Enterprise Java Beans. The EJB specification is currently undergoing major revisions in that area, as this proposal is being submitter. As a result, the submission team has chosen to use only those EJB security functions which can easily be mapped to CORBA security. Enhancing the security capability of EJB beyond the capabilities of CORBA security could impact, not only this submission, but CORBA security itself.

  Finally, this submission assumes a container will be built using a POA, most likely a **ServantLocator**. CORBA security depends on interceptors which are neither well-defined, portable between ORB implementations, or demonstratively capable of working with the POA. Fortunately, security policies can defined with the component deployment descriptor, enabling the component container to enforce authorization security by calling CORBA security operations directly, even if the security interceptor cannot.

2.2 Optional Requirements

- ***Responses may choose to specify enhancements to the standard CORBA Life Cycle Service that apply to components.***

  Enhancements to life cycle services are defined with the Components module (Chapter 5) and to the CosLifeCycle module (Chapter 14).

- ***Responses may choose to specify locality constraints for component management and construction. If an RFP for describing locality constraints is issued within the time frame of this RFP, responses to both RFPs shall be aligned***

  This submission introduces an new IDL construct, **local**, in Chapter 4 for use in defining locality-constrained interfaces. This construct is used to define all locality-constrained interfaces in this specification. Since the referenced RFP was never issued, no alignment is necessary.

Introduction to Components 3

This chapter is intended to provide a high-level introduction to the components submission. It will contain no specification material that is not defined in considerable more detail in the body of the submission. It is intended to introduce the key concepts before the reader delves into the next 300+ pages. Unfortunately it was not completed in time for this submission and consequently will be provided as an errata.

Extensions to CORBA Core 4

4.1 Local interface types

This specification provides a new CORBA meta-type that is used to define programming interfaces for locality-constrained objects. The syntax is similar to that of CORBA object interfaces, but the resulting type cannot be marshaled or remotely invoked. The *local* meta-type is intended to obviate the need for PIDL, to obviate the need for defining special "locality-constrained" cases of CORBA interfaces or abstract value types, and to provide users with a language-independent mechanism for declaring programming interfaces on local objects that leverages the CORBA typing system.

The grammar for specifying local interfaces is defined by the following BNF:

```
<local> ::= <local_header> "{" <local_member>* "}"

<local_header> ::= "local" [ <local_inheritance_spec> ]

<local_inheritance_spec> ::= ":"  <local_name>
                    { "," <local_name> } *

<local_member> ::= <local_op_dcl>
                    | <attr_dcl>
                    | <type_dcl>
                    | <const_dcl>
                    | <except_dcl>

<local_op_dcl> ::= <op_type_spec> <identifier> <parameter_dcls>
                    [ <raises_expr> ]

<local_name> ::= <scoped_name>

<local_base_type> ::= "localBase"
```

The semantics associated with local types are as follows:

- Local types cannot be marshaled. Consequently, local types (including sequences and arrays of local types) may not appear as parameters (or as components of any types that appear as parameters) of operations on CORBA Object interfaces. Local types (including sequences and arrays of local types) may not be members of structs, unions, or valuetypes. Local types may not be inserted into values of type **any**.
- Local types may appear as parameters or return values of operations on local types, or as attributes on local types.
- Parameters and return values of operations on local types may be any CORBA type. Attributes on local types may be any CORBA type.
- Language mappings for local types shall consist of the minimal language construct that satisfies the requirements of local types. In most object-oriented languages, it is expected that local types will be mapped to the language's fundamental object type, if one exists. The semantics of invocations on local types are the semantics of function or method calls in the underlying programming languages.
- When possible, language mappings for local types shall be syntactically similar to the mappings for interfaces. Inasmuch as possible, invocations on local types shall be consistent syntactically with invocations on CORBA objects with similar signatures.
- Language mappings shall specify the form of skeletons for local types to be generated by ORB products, allowing ORB users to provide implementations of local types. There is no specified generalized framework for managing the life cycles of user-defined local types (e.g., no standard factory mechanism). The life cycles of user-defined local types are determined by the life cycle constructs of underlying programming languages for base object types (e.g., constructors/destructors, garbage collection. etc.)

- Instances of local types have no inherent identities beyond their identities as programming objects. Specifically, there is no support for the concept of a reference to a local type, other than the basic programming language construct for referring to objects.
- Instances of local types defined as part of OMG specifications to be supplied by ORB products or object service products shall be exposed through the **ORB::resolve_local** operation or through some other local object obtained from **resolve_local**.
- The **localBase** keyword denotes the generalization of local types. When **localBase** is the formal type of a parameter in an operation, an instance of any specific local type may be passed as the actual parameter.
- Local types cannot be mapped to asynchronous invocation forms as specified by the CORBA Messaging Service specification.

4.1.1 Java language mapping

Local types map to the following Java constructs:

- A Java interface that corresponds to the specified local interface.
- A Holder class for the interface.
- If the local interface contains any type or constant definitions within its scope, a package corresponding to the interface scope.
- A Helper class

Applications may provide arbitrary classes that implement the mapped local interface.

The mapping for local interfaces is defined in terms of the mapping for normal Object interfaces. To determine the mapping for a given local interface, do the following:

1. Substitute the keyword **interface** for **local** in the IDL
2. Map the interface as currently specified by the IDL to Java language mapping (orbos/98-01-16).
3. The Java interface whose name is formed by appending the string “Operations” to the IDL interface name is identical to the mapping for the local interface.
4. If any types or constants are defined within the scope of the local interface scope, they are mapped exactly as they would be for a similar IDL Object interface. Their mapped types are placed in a package whose name is formed by appending the string “Package” to the interface name.
5. The holder class for a local interface is identical to an IDL Object interface of the same name.
6. The helper class for a local interface defines one public static method, id() which takes no parameters and returns a value of **org.omg.CORBA.RepositoryId**, which contains the repository ID of the local interface.

localBase maps to the **java.lang.Object** interface.

4.1.2 C++ language mapping

The C++ language mapping for local interfaces is almost identical to the mapping for abstract interfaces (ptc/98-09-03). Rather than defining a complete C++ mapping for abstract interfaces, which would only duplicate much of the specification of the mapping for abstract interfaces found in ptc/98-09-03, only the ways in which the local interface mapping differs from the abstract interface mapping are described here.

4.1.2.1 Local Interface Base

To avoid typing confusion, C++ classes for local interfaces are not derived from the **CORBA::AbstractBase** C++ class, but from a similar class called **CORBA::LocalBase**. As with **CORBA::AbstractBase, CORBA::LocalBase** facilitates narrowing and reference counting. All local interface base classes that have no other base local interfaces derive directly from **CORBA::LocalBase**. the **CORBA::LocalBase** C++ class is the mapping for the IDL **LocalBase** type. This base class provides the following:

- a protected default constructor
- a protected copy constructor
- a protected pure virtual destructor
- a public static **_duplicate** function
- a public static **_narrow** function
- a public static **_nil** function

The **LocalBase** class is shown below:

```
// C++
class LocalBase;
typedef LocalBase* LocalBase_ptr;
class LocalBase {
public:
static LocalBase_ptr _duplicate(LocalBase_ptr);
static LocalBase_ptr _narrow(LocalBase_ptr);
static LocalBase_ptr _nil();
protected:
LocalBase();
LocalBase(const LocalBase& val);
virtual ~LocalBase() = 0;
};
```

Local interface types support reference counting. The **LocalBase** class and the implementation of **release** overloaded for **LocalBase** are responsible for implementing reference counting. The **_duplicate** function increments the reference count of the argument and returns the argument. If the argument is a nil **LocalBase_ptr**, the return value is nil.

The implementation of **LocalBase::_narrow** is identical to that of **_duplicate**. **_duplicate** uses the value it returns as its own return value. Strictly speaking, the **_narrow** function is not needed in the **LocalBase** interface, but it is required by all conforming implementations so that **LocalBase** does not present a special case. As with regular object references, the **_nil** function returns a typed **LocalBase** nil reference.

Both the **is_nil** and **release** functions in the CORBA namespace are overloaded to handle local interface references:

```
// C++
namespace CORBA {
Boolean is_nil(LocalBase_ptr);
void release(LocalBase_ptr);
}
```

The **is_nil** function behaves identically to the other overloaded versions of the function.

If the argument to **release** is nil, then it does nothing. Otherwise, the it decrements the reference count on its argument. If the reference count is zero after being decremented, release destroys the argument.

4.1.2.2 Local interface mapping

The client side mapping for local interfaces is almost identical to the mapping for abstract interfaces, except:

- C++ classes for abstract interfaces derive from **CORBA::LocalBase**, not **CORBA::AbstractBase**.
- References to local interfaces cannot be inserted into a **CORBA::Any**
- Local interface references can only refer to local implementations of said interfaces. They may not refer to actual CORBA Objects or valuetypes.

Other than that, the mapping for abstract interfaces is identical to that for regular interfaces, including the following:

- a protected default constructor, a protected copy constructor, and a protected virtual destructor (because local interface classes server as base classes for application-supplied implementations)
- public virtual inheritance
- support for narrowing
- the provision of **_var** types, **_out** types
- the provision of manager types for struct, sequence, and array members
- identical memory management for parameters of operations
- identical C++ signatures for operations.

- operations and attribute accessors/mutators map to pure virtual functions

The application is responsible for providing a derived implementation of user-defined local interface types, and for implementing the proper behavior for memory management of parameter values.

4.1.3 resolve_local

This specification defines a new operation on the ORB pseudo-object that allows application programmers to obtain services expressed as local types. It is similar to **ORB::resolve_initial_references**, except that the operation return value is type **localBase**. The PIDL definition is as follows:

```
module CORBA {
// PIDL
interface ORB {
localBase resolve_local(in string name)
raises (InvalidName);
};
};
```

The string parameter to the **resolve_local** operation denotes a specific local object that is managed and supplied by the ORB or by services cooperating with the ORB. Specifications that define local interfaces that are not implemented by applications shall specify unique strings that will denote well-known local objects that can be obtained from **resolve_local**.

4.2 Import

This specification extends IDL to provide a mechanism for importing external name scopes into IDL specifications.

The grammar for the import statement is described by the following BNF:

<specification> ::= <import>* <definition>+

<import> ::= “import” <imported_scope> “;”

<imported_scope> ::= <scoped_name> | <string_literal>

The <*imported_scope*> non-terminal may be either a fully-qualified scoped name denoting an IDL name scope, or a string containing the interface repository ID of an IDL name scope, i.e., a definition object in the repository whose interface derives from **CORBA::Container**.

The definition of import obviates the need to define the meaning of IDL constructs in terms of “file scopes”. This specification defines the concepts of a *specification* as a unit of IDL expression. In the abstract, a *specification* consists of a finite sequence of

ISO Latin-1 characters that form a legal IDL sentence. The physical representation of the specification is of no consequence to the definition of IDL, though it is generally associated with a file in practice.

Any scoped name that begins with the scope token ("**::**") is resolved relative to the global scope of the specification in which it is defined. In isolation, the scope token represents the scope of the specification in which it occurs.

A specification that imports name scopes must be interpreted in the context a well-defined set of IDL specifications whose union constitutes the space from within which name scopes are imported. By "a well-defined set of IDL specifications", we mean any identifiable representation of IDL specifications, such as an interface repository. The specific representation from which name scopes are imported is not specified, nor is the means by which importing is implemented, nor is the means by which a particular set of IDL specifications (such as an interface repository) is associated with the context in which the importing specification is to be interpreted.

> *The above wording is deliberately imprecise. For example, we describe IDL specifications as being "interpreted in a particular context" rather than being compiled. Although IDL specifications exist most commonly as text files, and are usually processed by compilers, these are implementation artifacts that exist outside the scope of CORBA specifications. IDL specifications, from the perspective of CORBA specifications, are abstractions that may take an arbitrary number of forms, as long as they are unambiguously isomorphic to either a legal textual IDL specification or a legal construct in an interface repository. The use of a specification for a particular purpose (e.g., to generate stubs and skeletons) may be implemented in an arbitrary number of different ways, with or without compilers.*
>
> *In general, we expect that interface repositories will be a common means for supporting the import mechanism, and that compilers will be a common means for processing IDL specifications. In these cases, vendors will need to provide some means for users to associate the act of compilation with a particular interface repository, possibly through the use of environment variables or a system registry.*

The effects of an import statement are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in IDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- IDL module definitions may re-open modules defined in imported name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.

- When an name scope is imported, the names of the enclosing scopes in the fully-qualified pathname of the enclosing scope are *exposed* within the context of the importing specification, but their contents are not imported. An importing specification may not re-define or re-open a name scope which has been exposed (but not imported) by an import statement.
- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this specification, name scopes that can be imported (i.e., specified in an import statement) include the following: modules, interfaces, valuetypes, structures, unions, and exceptions.
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This specification does not define a particular form for generated stubs and skeletons in any given programming language. In particular, it does not imply any normative relationship between units specification and units of generation and/or compilation for any language mapping.

For example, assume that the following IDL has been processed and made available for importing by a particular product:

```
module A {
    struct outer {
        float f;
        string s;
    };
    interface I {
        struct inner {
            outer o;
            string s;
        };
    };
    interface J {
        exception badThing {};
    };
};

module B {
    typedef sequence<octet> mysteryBlob;
};
```

Consider the following specification in that context:

```
import ::A::I::inner;

import ::A::J;

import ::B;

module B {   // OK; re-opened
    interface K {
        void op1 (in ::A::I::inner val);  // OK
        void op2(in ::A::outer val);  // error; outer is not visible
        void op3(in long n) raises (::A::J::badThing); // OK
```

```
            void op4(in mysteryBlob blb);  // OK;
            // unqualified mysteryBlob resolves to imported B scope
        };
};

module A {   // error;
    // A is exposed, but not imported; it may not be re-opened
};
```

4.3 Repository identity declarations

This specification defines extensions to IDL to allow repository identifier values to be declared in a portable, standard manner. This mechanism is intended to obviate the **#pragma** mechanism currently specified (speaking in approximate terms) in section 10.6, “RepositoryIds”, of the CORBA 2.3 specification. Should this specification be adopted, the **#pragma** mechanisms shall be deprecated.

The following grammatical productions shall be added to the IDL grammar:

<type_id_dcl> ::= “typeId” <scoped_name> <string_literal>

<type_prefix_dcl> ::= “typePrefix” <scoped_name> <string_literal>

4.3.1 Repository identity declaration

The syntax of a repository identity declaration is as follows:

<type_id_dcl> ::= “typeId” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeId**
- a <*scoped_name*> that denotes the named IDL construct to which the repository identifier is assigned
- a string literal that must contain a valid repository identifier value

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface
- component
- home
- facet
- receptacle
- event sink
- event source
- finder

- factory
- value type
- value type member
- value box
- constant
- typedef
- exception
- attribute
- operation
- enum
- local

The value of the string literal is assigned as the repository identity of the specified type definition. This value will be returned as the RepositoryId by the interface repository definition object corresponding to the specified type definition. Language mappings constructs, such as Java helper classes, that return repository identifiers shall return the values declared for their corresponding definitions.

At most one repository identity declaration may occur for any named type definition. An attempt to re-define the repository identity for a type definition is illegal, regardless of the value of the re-definition.

If no explicit repository identity declaration exists for a type definition, the repository identifier for the type definition shall be an IDL format repository identifier, as defined in section 10.6.1 of the CORBA 2.3 specification.

4.3.2 Repository identifier prefix declaration

The syntax of a repository identifier prefix declaration is as follows:

<type_prefix_dcl> ::= “typePrefix” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeId**
- a <*scoped_name*> that denotes an IDL name scope to which the prefix applies
- a string literal that must contains the string to be pre-fixed to repository identifiers in the specified name scope

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface (including abstract interface)
- value type (including abstract, custom, and box value types)
- local interface
- specification scope (**::**)

The specified string is pre-fixed to the body of all repository identifiers in the specified name scope, whose values are assigned by default. To elaborate:

By “prefixed to the body of a repository identifier”, we mean that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon (“IDL:”) at the beginning of the identifier. A forward slash (‘/’) character is inserted between the end of the specified string and the remaining body of the repository identifier.

The prefix is only applied to repository identifiers whose values are not explicitly assigned by a typeId declaration. The prefix is applied to all such repository identifiers in the specified name scope, including the identifier of the construct that constitutes the name scope.

> *Note that this specification does not provide a mechanism that is analogous to the #pragma version mechanism. It is the considered opinion of the submitters that the current definition (or lack thereof) of the semantics of interface repository identifier versions is useless, or worse, misleading. To provide a mechanism for assigning so-called versions numbers would only invite further misuse.*

4.4 IDL Grammar modifications

In addition the extensions to IDL grammar specified in the previous sections, the following productions shall be modified to define the scopes in which local, typeId, and typePrefix may occur:

<definition> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <interface> “;”
| <module> “;”
| <value> “;”
| <local> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”

<export> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
|<op_dcl> “ ;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”

4.4.1 Keywords

This specification defines the following new keywords in IDL:

import local localBase typeId typePrefix

Component Model 5

5.1 Change History

The following changes have been made since the December 1998 version of the document (orbos./98-12-02) was posted:

1. Provide interfaces have been named **facets**.
2. The **home** declarations have been reworked to be compatible with the storage home construct of the persistence mechanism.
3. Miscellaneous clarifications have been made to the text.

All changes are clearly marked with change bars. In general existing text which was moved will not have change bars.

5.2 Component Model

Component is a new basic meta-type in CORBA. The component meta-type is an extension and specialization of the object meta-type. Component types can be specified in IDL and represented in the Interface Repository. A component is denoted by a component reference, which is a specialization of an object reference. Correspondingly, a component definition is a specialization and extension of an interface definition.

A component type is a specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository. Although the current specification does not attempt to provide mechanisms to support formal semantic descriptions associated with component definitions, our intent is that a component type definition is associated with a single well-defined set of behaviors. Although there may be several realizations of the component type for different run-time environments (e.g., OS/hardware platforms, languages, etc.), they should all

behave consistently. In this sense, a component type corresponds to an implementation type. As an abstraction in a type system, a component type is instantiated to create concrete entities (instances) with state and identity.

A component type encapsulates its internal representation and implementation; the "surface" of the component, as defined by its IDL description, is opaque from the perspective of the component's users. Although the component specification includes standard frameworks for component implementation, these frameworks, and any assumptions that they might entail, are completely hidden from clients of the component.

5.2.1 Ports

Components support a variety of surface features through which clients and other elements of an application environment may interact with a component. In general, these surface features are called *ports*. The component model supports four basic kinds of ports:

- **Facets**, which are distinct named interfaces provided by the component for client interaction
- **Receptacles**, which are named connection points that describe the component's ability to use a reference supplied by some external agent
- **Event sources**, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel
- **Event sinks**, which are named connection points into which events of a specified type may be pushed.
- **Attributes**, which are named values exposed through accessor and mutator operations. attributed are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

5.2.2 Components and facets

A component can provide multiple object references, called *facets*, which are capable of supporting distinct (i.e., unrelated by inheritance) CORBA interfaces. The component has a single distinguished reference whose interface conforms to the component definition. This reference supports an interface, called the component's *equivalent interface*, that manifests the component's surface features to clients. The equivalent interface allows clients to navigate among the component's facets, and to connect to the component's ports. The other interfaces provided by the component are referred to as *facets*. Figure 5-1 illustrates the relationship between the component and its facets.

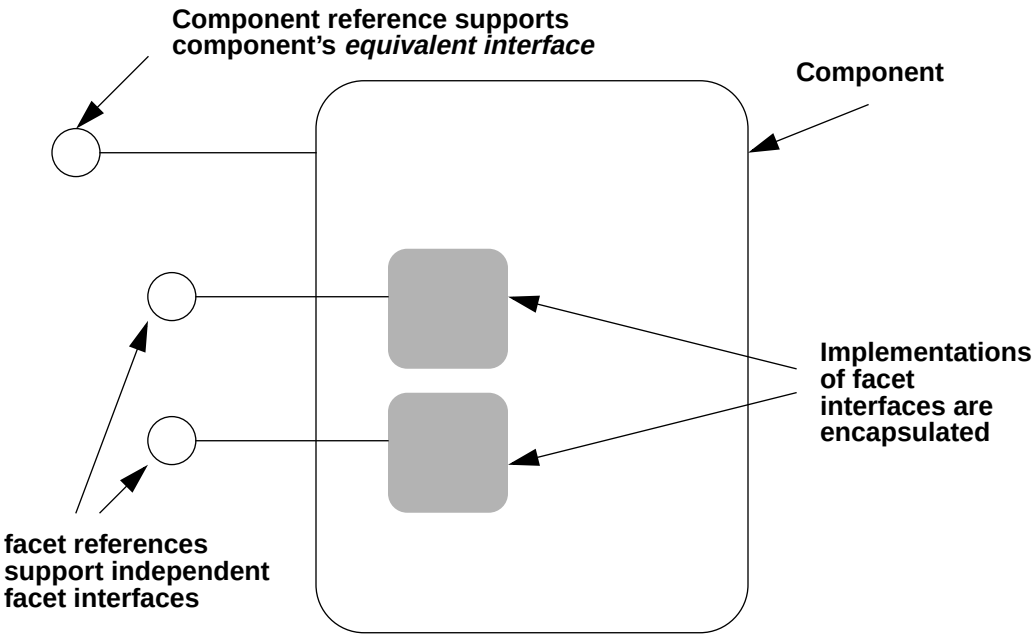

Figure 5-1 Component Interfaces and Facets

The relationship between the component and its facets is characterized by the following observations:

- The implementations of the facet interfaces are encapsulated by the component, and considered to be "parts" of the component. The internal structure of a component is opaque to clients.
- Clients can navigate from any facet to the component equivalent interface, and can obtain any facet from the component equivalent interface.
- Clients can reliably determine whether any two references belong to the same component instance.
- The life cycle of a facet reference is identical to the life cycle of its owning component.

5.2.3 Component identity

A component instance is identified primarily by its component reference, and secondarily by its set of facet references (if any). The component model provides operations to determine whether two references belong to the same component instance, and (as mentioned above) operations to navigate among a component's references. The definition of "same" component instance is ultimately up to the

component implementor, in that they may provide a customized implementation of this operation. However, the component framework provides standard implementations that constitute *de facto* definitions of “sameness” when they are employed.

Components may also be associated with *primary key values* by a component home. Primary keys are data values exposed to the component’s clients that may be used in the context of a component home to identify component instances and obtain references for them. Primary keys are not features of components themselves; the association between a component instance and a particular primary key value is maintained by the home that manages the component.

5.2.4 Component homes

This specification defines a *component home* meta-type that acts as a manager for instances of a specified component type. Component home interfaces provide operations to manage component life cycles, and optionally, to manage associations between component instances and primary key values. A component home may be thought of as a manager for the extent of a type (within the scope of a container).

Component types are defined in isolation, independent of home types. A home definition, however, must specify exactly one component type that it manages. Multiple different home types can manage the same component type, though they cannot manage the same set of component instances.

At execution time, a component instance is managed by a single home object of a particular type. The operations on the home are roughly equivalent to static or class methods in object-oriented programming languages.

5.3 Component Definition

5.3.1 IDL Extensions for Components

A component definition in IDL implicitly defines an interface that supports the features defined in the component definition body. It extends the concept of an interface definition to support features that are not supported in interfaces. Component definitions also differ from interface definitions in that they support only single inheritance from other component types.

The extensions to IDL for components are described by the following grammar.

```
<definition> ::= <type_dcl> ";"
                 | <const_dcl> ";"
                 | <except_dcl> ";"
                 | <interface> ";"
                 | <module> ";"
                 | <component> ";"
                 | <home_dcl> ";"

<component> ::= <component_dcl>
                 | <component_forward_dcl>

<component_forward_dcl> ::= "component" <identifier>

<component_dcl> ::= <component_header> "{" <component_body> "}"

<component_header> ::= "component" <identifier>
                 [ <component_inheritance_spec> ]
                 [ <supported_interface_spec> ]

<supported_interface_spec> ::= "supports" <scoped_name> { ","
                 <scoped_name> }*

<component_inheritance_spec> ::= ":" <scoped_name>

<component_body> ::= <component_export>*

<component_export> ::= <provides_dcl> ";"
                 | <uses_dcl> ";"
                 | <emits_dcl> ";"
                 | <publishes_dcl> ";"
                 | <consumes_dcl> ";"
                 | <attr_dcl>;

<provides_dcl> ::= "provides" <interface_type> <identifier>

<interface_type> ::= <scoped_name>
                 | "Object"

<uses_dcl> ::= "uses" [ "multiple" ] < interface_type> <identifier>

<emits_decl> ::= "emits" <scoped_name> <identifier>

<publishes_decl> ::= "publishes" <scoped_name> <identifier>

<consumes_dcl> ::= "consumes" <scoped_name> <identifier>

<attr_dcl> ::= <readonly_attr_spec>
                 | <attr_spec>

<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
                 <readonly_attr_declarator>
```

```
<readonly_attr_declarator> ::= <simple_declarator> [ <raises_expr> ]
                    | <simple_declarator> { “,” <simple_declarator> }*

<attr_spec> ::= “attribute” <param_type_spec> <attr_declarator>

<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
                    | <simple_declarator> { “,” <simple_declarator> }*

<attr_raises_expr> ::= <get_excep_expr> [ <set_excep_expr> ]
                    | <set_excep_expr>

<get_excep_expr> ::= “getRaises” <exception_list>

<set_excep_expr> ::= “setRaises” <exception_list>

<exception_list> ::= “(” <scoped_name> { “,” <scoped_name> } * “)”

<home_dcl> ::= <home_header> <home_body>

<home_header> ::= “home” <identifier> [ <home_inheritance_spec> ]
                    “manages” <scoped_name> [ <primary_key_spec> ]

<home_inheritance_spec> ::= “:” <scoped_name>

<primary_key_spec> ::= “primaryKey” <scoped_name>

<home_body> ::= “{” <home_export>* “}”

<home_export ::= <export>
                    | <factory_dcl> “;”
                    | <finder_dcl> “;”

<factory_dcl> ::= “factory” <identifier> “(“ [ <init_param_decls> ] “)” [
                    <raises_expr> ]

<finder_dcl> ::= “finder” <identifier> “(“ [ <init_param_decls> ] “)” [
                    <raises_expr> ]
```

5.4 Component Declaration

5.4.1 Syntax

The syntax for declaring a component header is as follows:

<component_dcl> ::= <component_header> “{” <component_body> “}”

<component_header> ::= “component” <identifier>
[<component_inheritance_spec>]
[<supported_interface_spec>]

<supported_interface_spec> ::= “supports” <scoped_name> { “,”
<scoped_name> }*

<component_inheritance_spec> ::= “:” <scoped_name>

A component header comprises the following elements:

- the keyword **component**
- an <*identifier*> that names the component type, and the equivalent interface, in the enclosing scope
- an optional <*inheritance_spec*>, consisting of a colon and a single <*scoped_name*> that must denote a previously-defined component type; see Section 5.13, “Component Inheritance” for details of component inheritance
- an optional <*supported_interface_spec*> that must denote one or more previously-defined IDL interfaces

5.4.2 Equivalent IDL

The client mappings (i.e., mappings of the externally-visible component features) for component declarations are described in terms of *equivalent IDL*. All of the features of components have equivalent forms in IDL as it exists are the time of this proposed specification (i.e., IDL grammar as specified by CORBA version 2.3).

As described above, the component meta-type is a specialization of the interface meta-type. Each component definition has a corresponding *equivalent interface*. In programming language mappings, components are denoted by object references that support the equivalent interface implied by the component definition.

5.4.2.1 Simple declaration

For a component declaration with the following form:

component *component_name* { ... };

the equivalent interface shall have the following form:

interface *component_name*
: Components::ComponentBase { ... };

5.4.2.2 Supported interfaces

For a component declaration with the following form:

component *<component_name>*
supports *<interface_name_1>, <interface_name_2>* { … };

the equivalent interface shall have the following form:

interface <*component_name*>
: Components::ComponentBase,
***<interface_name_1>, <interface_name_2>* { … };**

Supported interfaces are described in detail in Section 5.5.6 on page 44

5.4.2.3 Inheritance

For a component declaration with the following form:

component <*component_name*> : *<base_name>* { … };

the equivalent interface shall have the following form:

interface <*component_name*> : *<base_name>* { … }

5.4.2.4 Inheritance and supported interfaces

For a component declaration with the following form:

component *<component_name> : <base_name>*
supports *<interface_name_1>, <interface_name_2>* { … };

the equivalent interface shall have the following form:

interface *<component_name>*
: *<base_name>, <interface_name_1>, <interface_name_2>* { … };

5.4.3 Component Body

A component forms a naming scope, nested within the scope in which the component is declared. A component body can contain the following kinds of port declarations:

- Provided interface declarations (**provides**)
- Receptacle declarations (**uses**)
- Event source declarations (**emits** or **publishes**)
- Event sink declarations (**consumes**)
- Attribute declarations (**attribute**)

Declarations for facets, receptacles, events sources, event sinks and attributes all map onto operations on the component's equivalent interface. These declarations and their meanings are described in detail below.

5.5 Facets and Navigation

A component type may provide several independent interfaces to its clients in the form of facets. Facets are intended to be the primary vehicle through which a component exposes its functional application behavior to clients during normal execution.

5.5.1 Syntax

A facet is declared with the following syntax:

<provides_dcl> ::= “provides” <interface_type> <identifier>

The interface type must be either the keyword **Object**, or a scoped name that denotes a previously-declared interface type which is not a component interface, i.e., is not the interface corresponding to a component definition. The identifier names the facet within the scope of the component, allowing multiple facets of the same type to be provided by the component.

5.5.2 Equivalent IDL

Facet declarations imply operations on the component interface that provide access to the provided interfaces by their names. A facet declaration of the following form:

provides *<interface_type>* *<name>*;

results in the following equivalent operation defined in the component interface:

***<interface_type>* provide_*<name>* ();**

The mechanisms for navigating among a component’s facets are described in section Section 5.5.4 on page 40. The relationships between the component identity and the facet references, and assumptions regarding facet references, are described in section Section 5.5.5 on page 44. The implementation of navigation operations are provided by the component implementation framework in generated code; the user-provided implementation of a component type is not responsible for navigation operations. The responsibilities of the component servant framework for supporting navigation operations are described in detail in Chapter 6.

5.5.3 Semantics of facet references

Clients of a component instance can obtain a reference to a facet by invoking the **provide_*<name>*** operation on the component interface corresponding to the **provides** declaration in the component definition. The component implementation is responsible for guaranteeing the following behaviors:

- In general, a component instance should be prepared to return object references for facets throughout the instance’s life cycle. A component implementation may, as part of its advertised behavior, return a nil object reference as the result of a **provide_*<name>*** operation.

- An object reference returned by a **provide_*<name>*** operation must support the interface associated with the corresponding **provides** declaration in the component definition. Specifically, when the **_is_a** operation is invoked on the object reference with the **RepositoryId** of the provided interface type, the result must be **true**, and legal operations of the facet interface must be able to be invoked on the object reference. If the type specified in the **provides** declaration is **Object**, then there are no constraints on the interface types supported by the reference.

  A facet reference provided by a component may support additional interfaces, such as interfaces derived from the declared type, as long as the stated contract is satisfied.

- Facet references must behave properly with respect to component identity and navigation, as defined in sections Section 5.5.4 on page 40.

5.5.4 Navigation

Navigation among a component's facets may be accomplished in the following ways:

- A client may navigate from any facet reference to the component that provides the reference via **CORBA::Object::get_component**.
- A client may navigate from the component interface to any facet using the generated **provide_*<name>*** operations on the component interface.
- A client may navigate from the component interface to any facet using the generic **provide_facet** operation on the **Navigation** interface (inherited by all component interfaces through **Components::ComponentBase**). Other operations on the **Navigation** interface (i.e., **provide_all_facets** and **provide_named_facets**) return multiple references, and can also be used for navigation. When using generic navigation operations on **Navigation**, facets are identified by string values that contain their declared names.
- A client may navigate from a facet interface that derives from the **Navigation** interface directly to any other facet on the same component, using **provide_facet**, **provide_all_facets**, and **provide_named_facets**.

The detailed descriptions of these mechanisms follow.

5.5.4.1 get_component()

The CORBA component specification extends the **CORBA::Object** pseudo interface with a single operation:

```
module CORBA {
    interface Object { // PIDL
        ...
        Object get_component ( );
    };
};
```

If the target object reference is itself a component reference (i.e., it denotes the component itself), the **get_component** operation returns the same reference (or another equivalent reference). If the target object reference is a facet reference the **get_component** operation returns an object reference for the component. If the target reference is neither a component reference nor a provided reference, **get_component** returns a nil reference.

Implementation of get_component

As with other operations on **CORBA::Object**, **get_component** is implemented as a request to the target object. Following the pattern of other **CORBA::Object** operations (i.e., **_interface**, **_is_a**, and **_non_existent**; see section 15.4.1.2 of the CORBA 2.3 specification), the operation name in GIOP request corresponding to **get_component** shall be "**_component**".

Programming skeletons generated by the Component Implementation Framework for components and facets shall provide standard implementations for **get_component** (i.e., the **_component** request).

5.5.4.2 *Component-specific* provide *operations*

The **provide_*<name>*** operation implicitly defined by a **provides** declaration can be invoked to obtain a reference to the facet.

5.5.4.3 *Navigation interface on the component*

As described in Section 5.4 on page 36 all component interfaces implicitly inherit directly or indirectly from **ComponentBase**, which inherits from **Components::Navigation**. The definition of the **Components::Navigation** interface is as follows:

```
module Components {

    interface Navigation {

        valuetype FacetDescription {
            public RepositoryID InterfaceID;
            public FeatureName Name;
        };

        valuetype Facet : FacetDescription {
            public Object Ref;
        };

        typedef sequence<Facet> Facets;

        typedef sequence<FacetDescription>
            FacetDescriptions;

        Object provide_facet( in FeatureName name )
            raises (InvalidName);

        FacetDescriptions describe_facets();

        Facets provide_all_facets();

        Facets provide_named_facets(in NameList names)
            raises (InvalidName);

        boolean same_component( in Object ref );

    };
};
```

This interface provides generic navigation capabilities. It is inherited by all component interfaces, and may be optionally inherited by any interface that is explicitly designed to be a facet interface for a component. The descriptions of **Navigation** operations follow.

provide_facet

The **provide_facet** operation returns a reference to the facet denoted by the **name** parameter. The value of the **name** parameter must be identical to the name specified in the provides declaration. The valid names are defined by inherited closure of the actual type of the component, i.e., the names of facets of the component type and all of its inherited component types. If the value of the **name** parameter does not correspond to one of the component's facets, the **InvalidName** exception is raised.

describe_facets

The **describe_facets** operation returns a sequence containing descriptions of all of the facets provided by the target component. Each description is a value type containing the **RepositoryId** of the facet's interface and the name of the facet, expressed as an unscoped local name relative to the owning component's name scope. The order in which these descriptions occur in the sequence is not specified.

provide_all_facets

The **provide_all_facets** operation returns a sequence of value objects, each of which contains the **RepositoryId** of the facet interface and **name** of the facet, along with a reference to the facet. The sequence must contain descriptions and references for all of the facets in the component's inheritance hierarchy. The order in which these values occur in the sequence is not specified.

provide_named_facets

The **provide_named_facets** operation returns a sequence of described references (identical to the sequence returned by **provide_all_facets**), containing descriptions and references for the facets denoted by the **names** parameter. If any name in the **names** parameter is not a valid name for a provided interface on the component, the operation raises the **InvalidName** exception. The order of values in the returned sequence is not specified.

The **same_component** operation on **Navigation** is described in Section 5.5.5 on page 44.

5.5.4.4 Navigation interface on facet interfaces

Any interface that is designed to be used as a facet interface on a component may optionally inherit from the **Navigation** interface. When the navigation operations (i.e., **provide_facet**, **provide_all_facets**, **provide_named_facets**, and **describe_facets**) are invoked on the facet reference, the operations shall return the same results as if they had been invoked on the component interface that provided the target facet. The skeletons generated by the Component Implementation Framework will provide implementations of these operations that will delegate to the component interface.

This option allows navigation from one facet to another to be performed in a single request, rather than a pair of requests (to get the component reference and navigate from there to the desired facet). To illustrate, consider the following component definition:

```
module example {
    interface foo : Components::Navigation {... };
    interface bar { ... };
    component baz session {
        provides foo a;
        provides bar b;
    };
};
```

A client could navigate from a to b as follows:

```
foo myFoo;
// assume myFoo holds a reference to a foo provided by a baz
baz myBaz = bazHelper.narrow(myFoo.get_component());
bar myBar = myBaz.provide_b();
```

Or, it could navigate directly:

```
foo myFoo;
// assume myFoo holds a reference to a foo provided by a baz
bar myBar = barHelper.narrow(myFoo.provide_interface("b");
```

5.5.5 Provided References and Component Identity

The **same_component** operation on the **Navigation** interface allows clients to determine reliably whether two references belong to the same component instance, that is, whether the references are facets of or directly denote the same component instance. The component implementation is ultimately responsible for determining what the "same component instance" means. The skeletons generated by the Component Implementation Framework provide an implementation of **same_component**, where "same instance" is defined in terms of opaque identity values supplied by the component implementation or the container in the container context. User-supplied implementations can provide different semantics.

If a facet interface inherits the **Navigation** interface, then the **same_component** operation on the provided interface should give the same results as the **same_component** operation on the component interface that owns the provided interface. The skeletons generated by the Component Implementation Framework provide an implementation of **same_component** for facets that inherit the **Navigation** interface.

5.5.6 Supported interfaces

A component definition may optionally support one or more interfaces, or inherit from a component that supports one or more interfaces. When a component definition header includes a supports clause as follows:

component <*component_name*> supports <*interface_name*> { … };

the equivalent interface inherits both **ComponentBase** and any supported interfaces, as follows:

interface <*component_name*>
: Components::ComponentBase, <*interface_name*> { … };

The component implementation must supply implementations of operations defined on supported interfaces. Clients must be able to widen a reference of the component's equivalent interface type to the type of any of the supported interfaces. Clients must also be able to narrow a reference of type **ComponentBase** to the type of any of the component's supported interfaces.

For example, given the following IDL:

```
module M {
    interface I {
        void op();
    };
    component A supports I {
        provides I foo;
    };
    home AManager manages A { };
};
```

The AManager interface will be derived from KeylessHomeBase, supporting the create_component operation, which returns a reference of type ComponentBase. This reference must be able to be narrowed directly from ComponentBase to I:

```
// java
...
M.AManager aHome = ...; // get A's home
org.omg.Components.ComponentBase myComp =
aHome.create_component();
M.I myI = M.IHelper.narrow(myComp);
// must succeed
```

For example, given the following IDL:

```
module M {
    interface I {
        void op();
    };
    component A supports I {
        provides I foo;
    };
    component B : A { ... };
    home BHome manages B {};
};
```

The equivalent IDL is:

```
module M {
    interface I {
        void op();
    };
    interface A :
    org.omg.Components.ComponentBase, I { ... };
    interface B : A { ... };
};
```

which allows the following usage:

```
M.BHome bHome = ... // get B's home
M.B myB = bHome.create();
myB.op();          // I's operations are supported
                   // directly on B's interface
```

The supports mechanism provides programming convenience for lightweight components that only need to implement a single operational inter-

face. A client can invoke operations from the supported interface directly on the component reference, without narrowing or navigation:

```
M.A myA = aHome.create();
myA.op();
```

as opposed to

```
M.A myA = aHome.create();
M.I myI = myA.provide_foo();
myI.op();
```

or, assuming that the client has A's home, but doesn't statically know about A's interface or home interface:

```
org.omg.Components.KeylessHomeBase genericHome =
... // get A's home;
org.omg.Components.ComponentBase myComp =
genericHome.create_component();

M.I myI = M.IHelper.narrow(myComp);
myI.op();
```

as opposed to

```
org.omg.CORBA.Object obj =
myComp.provide_interface("foo");
M.I myI = M.IHelper.narrow(obj);
myI.op();
```

This mechanism allows component-unaware clients to receive a reference to a component (passed as type CORBA::Object) and use the supported interface.

5.6 Receptacles

A component definition can describe the ability to accept object references upon which the component may invoke operations. When a component accepts an object reference in this manner, the relationship between the component and the referent object is called a *connection*; they are said to be *connected*. The conceptual point of connection is called a *receptacle*. A receptacle is an abstraction that is concretely manifested on a component as a set of operations for establishing and managing connections.

Receptacles are intended as a mechanical device for expressing a wide variety of relationships that may exist at higher levels of abstraction. As such, receptacles have no inherent higher-order semantics, such as implying ownership, or that certain operations will be transient across connections.

5.6.1 Syntax

The syntax for describing a receptacle is as follows:

<uses_dcl> ::= “uses” [“multiple”] <interface_type> <identifier>

A receptacle declaration comprises the following elements:

- The keyword **uses**.
- The optional keyword **multiple**. The presence of this keyword indicates that the receptacle may accept multiple object references simultaneously, and results in different operations on the component’s associated interface.
- An <*interface_type*>, which must be either the keyword **Object** or a scoped name that denotes the interface type that the receptacle will accept. The scoped name must denote a previously-defined non-component interface type.
- An <*identifier*> that names the receptacle in the scope of the component.

5.6.2 Equivalent IDL

A **uses** declaration of the following form:

uses *<interface_type> <receptacle_name>*;

results in the following equivalent operations defined in the component interface:

```
void connect_<receptacle_name> ( in <interface_type> conxn )
raises (
    Components::AlreadyConnected,
    Components::InvalidConnection
);

<interface_type> disconnect_<receptacle_name> ( )
raises ( Components::NoConnection );

<interface_type> get_connection_<receptacle_name> ( );
```

A **uses** declaration of the following form:

uses multiple *<interface_type> <receptacle_name>*;

results in the following equivalent operations defined in the component interface:

```
struct <receptacle_name>Connection {
    <interface_type> objref;
    Components::Cookie ck;
};
sequence< <receptacle_name>Connection>
<receptacle_name>Connections;

Components::Cookie
connect_<receptacle_name> ( in <interface_type> connection )
raises (
    Components::ExceededConnectionLimit,
    Components::InvalidConnection
);

<interface_type> disconnect_<receptacle_name> (
    in Components::Cookie ck
)
raises ( Components::InvalidConnection );

<receptacle_name>Connections
get_connections_<receptacle_name> ( );
```

5.6.3 Behavior

5.6.3.1 Connect operations

Operations of the form **connect_*<receptacle_name>*** are implemented in part by the component implementor, and in part by generated code in the component servant framework. The responsibilities of the component implementation and servant framework for implementing connect operations are described in detail in Chapter 6. The receptacle holds a copy of the object reference passed as a parameter. The component may invoke operations on this reference according to its design. How and when the component invokes operations on the reference is entirely the prerogative of the component implementation. The receptacle will hold a copy of the reference until it is explicitly disconnected.

Simplex receptacles

If a receptacle's **uses** declaration does not include the optional **multiple** keyword, then only a single connection to the receptacle may exist at a given time. If a client invokes a connect operation when a connection already exists, the connection operation will raise the **AlreadyConnected** exception.

The component implementation may refuse to accept the connection for arbitrary reasons. If it does so, the connection operation will raise the **InvalidConnection** exception.

Multiplex receptacles

If a receptacle's **uses** declaration includes the optional **multiple** keyword, then multiple connections to the receptacle may exist simultaneously. The component implementation may choose to establish a limit on the number of simultaneous connections allowed. If an invocation of a connect operation attempts to exceed this limit, the operation will raise the **ExceededConnectionLimit** exception.

The component implementation may refuse to accept the connection for arbitrary reasons. If it does so, the connection operation will raise the **InvalidConnection** exception.

Connect operations for multiplex receptacles return values of type **Components::Cookie**. Cookie values are used to identify the connection for subsequent disconnect operations. Cookie values are generated by the receptacle implementation (the responsibility of the supplier of the component-enabled ORB, not the component implementor). Likewise, cookie equivalence is determined by the implementation of the receptacle implementation.

The client invoking connection operations is responsible for retaining cookie values and properly associating them with connected object references, if the client needs to subsequently disconnect specific references. Cookie values must be unique within the scope of the receptacle that created them. If a cookie value is passed to a disconnect operation on a different receptacle than that which created it, results are undefined.

Cookie values are described in detail in Section 5.6.3.4, "Cookie type".

Cookie values are required because object references cannot be reliably tested for equivalence.

5.6.3.2 Disconnect operations

Operations of the form **disconnect_*receptacle_name*** terminate the relationship between the component and the connected object reference.

Simplex receptacles

If a connection exists, the disconnect operation will return the connected object reference. If no connection exists, the operation will raise a **NoConnectionExists** exception.

Multiplex receptacles

The **disconnect_*receptacle_name*** operation of a multiplex receptacle takes a parameter of type **Components::Cookie**. The **ck** parameter must be a value previously returned by the **connect_*receptacle_name*** operation on the same receptacle. It is the responsibility of the client to associate cookies with object references they connect and disconnect. If the cookie value is not recognized by the receptacle implementation as being associated with an existing connection, the **disconnect_*receptacle_name*** operation raises an **InvalidConnection** exception.

5.6.3.3 get_connection and get_connections operations

Simplex receptacles

Simplex receptacles have operations named **get_connection_*receptacle_name***. If the receptacle is currently connected, this operation returns the connected object reference. If there is no current connection, the operation returns a nil object reference.

Multiplex receptacles

Multiplex receptacles have operations named **get_connections_*receptacle_name***. This operation returns a sequence of structures, where each structure contains a connected object reference and its associated cookie value. The sequence contains a description of all of the connections that exist at the time of the invocation. If there are no connections, the sequence length will be zero.

5.6.3.4 Cookie type

The **Cookie** valuetype is defined by the following IDL:

```
module Components {

    valuetype Cookie {
        private sequence<octet> cookieValue;
    };
};
```

Cookie values are created by multiplex receptacles, and are used to correlate a connect operation with a disconnect operation on multiplex receptacles.

Implementations of component-enabled ORBs may employ value type derived from **Cookie**, but any derived cookie types must be truncatable to **Cookie**, and the information preserved in the **cookieValue** octet sequence must be sufficient for the receptacle implementation to identify the cookie and its associated connected reference.

5.6.4 Receptacles interface

The **Receptacles** interface provides generic operations for connecting to a component's receptacles. The **ComponentBase** interface is derived from **Receptacles**. The **Receptacles** interfaces is defined by the following IDL:

```
module Components {

    valuetype ConnectionDescription {
        public Cookie ck;
        public Object objref;
    };

    typedef sequence<ConnectionDescription> ConnectedDescriptions;

    interface Receptacles {

        Cookie connect (
            in FeatureName name,
            in Object connection )
        raises (
            InvalidName,
            InvalidConnection,
            AlreadyConnected,
            ExceededConnectionLimit);

        void disconnect (
            in FeatureName name,
            in Cookie ck)
        raises (
            InvalidName,
            InvalidConnection,
            CookieRequired,
            NoConnection);

        ConnectionList get_connections (in FeatureName name)
        raises (InvalidName);

    };
};
```

connect

The **connect** operation connects the object reference specified by the **connection** parameter to the receptacle specified by the **name** parameter on the target component. If the specified receptacle is a multiplex receptacle, the operation returns a cookie value that can be used subsequently to disconnect the object reference. If the receptacle is a simplex receptacle, the return value is a nil. The following exceptions may be raised:

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle and it is already connected, then the **AlreadyConnected** exception is raised.

- If the object reference in the **connection** parameter does not support the interface declared in the receptacle's **uses** statement, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the implementation-defined limit to the number of connections is exceeded, the **ExceededConnectionLimit** exception is raised.

disconnect

If the receptacle identified by the **name** parameter is a simplex receptacle, the operation will disassociate any object reference currently connected to the receptacle. The cookie value in the **ck** parameter is ignored. If the receptacle identified by the **name** parameter is a multiplex receptacle, the **disconnect** operation disassociates the object reference associated with the cookie value (i.e., the object reference that was connected by the operation that created the cookie value) from the receptacle. The following exceptions may be raised:

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle there is no current connection, then the **NoConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the cookie value in the **ck** parameter does not denote an existing connection on the receptacle, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and a null value is specified in the **ck** parameter, the **CookieRequired** exception is raised.

get_connections

The **get_connections** operation returns a sequence of **ConnectionDescription** structs. Each struct contains an object reference connected to the receptacle named in the **name** parameter, and a cookie value that denotes the connection. If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.

5.7 Events

The CORBA component model supports a publish/subscribe event model. The event model for CORBA components is designed to be compatible with the OMG Event and Notification Services, as defined in OMG documents formal/97-12-11 and telcom/98-11-01, but it does not require that either service be used to implement the component event model. The interfaces exposed by the component event model provide a simple programming interface whose semantics can be mapped onto a subset of Event and Notification Service semantics.

5.7.1 Event types

Event types in the CORBA Component event model are value types derived from the abstract value type **Components::EventBase**, which is defined as follows:

```
module Components {
    abstract valuetype EventBase {};
};
```

Applications derive specific concrete event types from this base type.

When the underlying implementation of the component event mechanism provided by the container is either the CORBA Event Service or the CORBA Notification Service, event values shall be inserted into instances of the **any** type. The resulting **any** values can be used as parameters to the push operation on untyped event channels, or inserted into a structured event for use with the Notification Service.

5.7.2 Integrity of value types contained in anys

To ensure proper transmission of value type events, this specification makes the following clarifications to the semantics of value types when inserted into **any**s:

When an **any** containing a value type is received as a parameter in an ORB-mediated operation, the value contained in the **any** must be preserved, regardless of whether the receiving execution context is capable of constructing the value (in its original form or a truncated form), or not. If the receiving context attempts to extract the value, the extraction may fail, or the extracted value may be truncated. The value contained in the **any** shall remain unchanged, and shall retain its integrity if the **any** is passed as a parameter to another execution context.

5.7.3 EventConsumer interface

The component event model is a push model. The basic mechanics of this push model are defined by consumer interfaces. Event sources hold references to consumer interfaces and invoke various forms of push operations to send events. Component event sources hold references to consumer interfaces and push to them. Component event sinks provide consumer references, into which other entities (e.g., channels, clients, other component event sources) push events.

Event consumer interfaces are derived from the **Components::EventConsumerBase** interface, which is defined as follows:

```
module Components {
    exception BadEventType {
        CORBA::RepositoryId expected_event_type
    };
    interface EventConsumerBase {
        void push_event(in EventBase evt) raises (BadEventType);
    };
};
```

Type-specific event consumer interfaces are derived from the **EventConsumerBase** interface. Event source and sink declarations in component definitions cause type-specific consumer interfaces to be generated for the event types used in the declarations.

The **push_event** operation pushes the event denoted by the **evt** parameter to the consumer. The consumer may choose to constrain the type of event it accepts. If the actual type of the **evt** parameter is not acceptable to the consumer, the **BadEventType** exception is raised. The **expected_event_type** member of the exception contains the repository ID of the type expected by the consumer.

Note that this exception can only be raised by the consumer upon whose reference the **push_event** operation was invoked. The consumer may be a proxy for an event or notification channel with an arbitrary number of subscribers. If any of those subscribers raise any exceptions, they will not be propagated back to the original event source (i.e., the component).

5.7.4 Event service provided by container

Container implementations provide event services to components and their clients. Component implementations obtain event services from the container during initialization, and mediate client access to those event services. The container implementation is free to provide any mechanism that supports the required semantics. The container is responsible for configuring the mechanism and determining the specific quality of service and routing policies to be employed when delivering events. More detail is defined in Chapter 8, specifically Section 8.5 on page 221.

5.7.5 Event Sources—publishers and emitters

An event source embodies the potential for the component to generate events of a specified type, and provides mechanisms for associate consumers with sources.

There are two categories of event sources, *emitters* and *publishers*. An emitter can be connected to at most one consumer. A publisher can be connected to an arbitrary number of consumers, who are said to *subscribe* to the publisher event source.

A *publisher* event source has the following characteristics:

- The equivalent operations for publishers allow multiple subscribers (i.e., consumers) to connect to the same source simultaneously.

- Subscriptions to a publisher are delegated to an event channel supplied by the container at run time. The component is guaranteed to be the only source publishing to that event channel.

An *emitter* event source has the following characteristics:

- The equivalent operations for emitters allow only one consumer to be connected to the emitter at a time.
- The events pushed from an emitter to the connected consumer are not mediated by a channel associated with the component or the emitter. Events pushed from an emitter are pushed directly into to consumer interface supplied as a parameter to the connect_<*source*> operation.

> *In general, emitters are not intended to be exposed to clients. Rather, they are intended to be used for configuration purposes. It is expected that emitters will be connected at the time of component initialization and configuration to consumer interfaces that are proxies for event channels that may be shared between arbitrary clients, components and other system elements.*
>
> *In contrast, publishers are intended to provide clients with direct access to a particular event stream being generated by the component (embodied by the publisher event source). It is our intent that clients subscribe directly to the publisher source.*

5.7.6 Publisher

5.7.6.1 Syntax

The syntax for an event publisher is as follows:

<publishes_decl> ::= “publishes” <scoped_name> <identifier>

A publisher declaration consists of the following elements:

- the keyword **publishes**
- a <*scoped_name*> that denotes a previously-defined value type derived from **Components::EventBase**
- an <*identifier*> that names the publisher event source in the scope of the component

5.7.6.2 Equivalent IDL

For an event source declaration of the following form:

```
module <module_name> {
    component <component_name> {
        publishes <event_type> <source_name>;
    };
};
```

The following equivalent IDL is implied:

```
module <module_name> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

    interface <component_name> : Components::ComponentBase {

        Components::Cookie subscribe_<source_name>(
            in
            <component_name>EventConsumers::<event_type>Consumer
            consumer
        )
        raises (
            Components::ExceededConnectionLimit
        );

        <component_name>EventConsumers::<event_type>Consumer
        unsubscribe_<source_name> (in Components::Cookie ck)
        raises (Components::InvalidConnection);
    };

    module <component_name>EventConsumers {
        interface <event_type>Consumer
        : Components::EventConsumerBase {
            void push(in <event_type> evt);
        };
    };
};
```

5.7.6.3 Event publisher operations

subscribe_*<source_name>*

The **subscribe_*<source_name>*** operation connects the consumer parameter to an event channel provided to the component implementation by the container. The component will be the only publisher to that channel. If the implementation of the component or the channel place an arbitrary limit on the number of subscriptions that can be supported simultaneously, and the invocation of the subscribe operation would cause that limit to be exceeded, the operation raises the **ExceededConnectionLimit** exception.The Cookie value returned by the operation identifies the subscription

formed by the association of the subscriber with the publisher event source. This value can be used subsequently in an invocation of **unsubscribe_*<source_name>*** to disassociate the subscriber from the publisher.

unsubscribe_<source_name>

The **unsubscribe_*<source_name>*** operation destroys the subscription identified by the **ck** parameter value, returning the reference to the subscriber. If the **ck** parameter value does not identify an existing subscription to the publisher event source, the operation raises a **InvalidConnection** exception.

5.7.7 Emitters

5.7.7.1 Syntax

The syntax for an emitter declaration is as follows:

<emits_decl> ::= “emits” <scoped_name> <identifier>

An emitter declaration consists of the following elements:

- the keyword **emits**
- a <*scoped_name*> that denotes a previously-defined value type derived from **Components::EventBase**
- an <*identifier*> that names the event source in the scope of the component

5.7.7.2 Equivalent IDL

For an event source declaration of the following form:

```
module <module_name> {
    component <component_name> {
        emits <event_type> <source_name>;
    };
};
```

The following equivalent IDL is implied:

```
module <module_name> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

    interface <component_name> : Components::ComponentBase {

        void connect_<source_name>(
            in
            <component_name>EventConsumers::<event_type>Consumer
            consumer
        )
        raises (
            Components::AlreadyConnected
        );

        <component_name>EventConsumers::<event_type>Consumer
        disconnect_<source_name>()
        raises (Components::NoConnection);
    };

    module <component_name>EventConsumers {
        interface <event_type>Consumer
        : Components::EventConsumerBase {
            void push(in <event_type> evt);
        };
    };
};
```

5.7.7.3 Event emitter operations

connect_*<source_name>*

The **connect_*<source_name>*** operation connects the event consumer denoted by the consumer parameter to the event emitter. If the emitter is already connected to a consumer, the operation raises the **AlreadyConnected** exception.

disconnect_*<source_name>*

The **disconnect_*<source_name>*** operation destroys any existing connection by disassociating the consumer from the emitter. The reference to the previously connected consumer is returned. If there was no existing connection, the operation raises the **NoConnection** exception.

5.7.8 Module scope of generated event consumer interfaces

The following observations and constraints apply to the equivalent IDL for event source declarations:

- The need for a typed event consumer interface requires the definition of a module scope to guarantee that the interface name for the event subscriber is unique. The module (whose name is formed by appending the string “EventConsumers” to the component type name) is defined in the same scope as the component’s equivalent interface. The module is opened before the equivalent interface definition to provide forward declarations for consumer interfaces. It is re-opened after the equivalent interface definition to define the consumer interfaces.
- The name of a consumer interface is formed by appending the string “Consumer” to the name of the event type. One consumer interface type is implied for each unique event type used in event source and event sink declarations in the component definition.

5.7.9 Event Sinks

An event sink embodies the potential for the component to receive events of a specified type. An event sink is, in essence, a special-purpose facet whose type is an event consumer. External entities, such as clients or configuration services, can obtain the reference for the consumer interface associated with the sink.

Unlike event sources, event sinks do not distinguish between *connection* and *subscription*. The consumer interface may be associated with an arbitrary number of event sources, unbeknownst to the component that supplies the event sink. The component event model provides no inherent mechanism for the component to control which events sources may be pushing to its sinks. By exporting an event sink, the component is, in effect, declaring its willingness to accept events pushed from arbitrary sources.

> *If a component implementation needs control over which sources can push to a particular sink it owns, the sink should not be exposed as a port on the component. Rather, the component implementation can create a consumer internally and explicitly connect or subscribe it to sources.*

5.7.9.1 Syntax

The syntax for an event sink declaration is as follows:

<consumes_dcl> ::= “consumes” <scoped_name> <identifier>

An event sink declaration contains the following elements:

- the keyword **consumes**
- a <*scoped_name*> that denotes a previously-defined value type that is derived from the Components::EventBase abstract value type
- an <*identifier*> that names the event sink in the component’s scope

5.7.9.2 Equivalent IDL

For an event sink declaration of the following form:

```
module <module_name> {
    component <component_name> {
        consumes <event_type> <sink_name>;
    };
};
```

The following equivalent IDL is implied:

```
module <module_name> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

    interface <component_name> : Components::ComponentBase {
        <component_name>EventConsumers::<event_type>Consumer
        get_consumer_<sink_name>();
    };

    module <component_name>EventConsumers {
        interface <event_type>Consumer
        : Components::EventConsumerBase {
            void push(in <event_type> evt);
        };
    };
};
```

5.7.9.3 Event sink operations

The **get_consumer_*<sink_name>*** operation returns a reference that supports the consumer interface specific to the declared event type.

5.7.10 Events interface

The **Events** interface provides generic access to event sources and sinks on a component. **ComponentBase** is derived from **Events**. The **Events** interface is described as follows:

```
module Components {

    interface Events {
        EventConsumerBase
        get_consumer(in FeatureName sinkName)
            raises(InvalidName);
        Cookie subscribe(in FeatureName publisherName,
            in EventConsumerBase subscriber)
            raises (InvalidName);
        void unsubscribe(in FeatureName publisherName,
            in Cookie ck)
            raises(InvalidName, InvalidConnection);
        void connect_consumer(in FeatureName emitterName,
            in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected);
        EventConsumerBase
        disconnect_consumer(in FeatureName sourceName)
            raises(InvalidName, NoConnection);

    };

};
```

get_consumer

The **get_consumer** operation returns the **EventConsumerBase** interface for the sink specified by the **sinkName** parameter. If the **sinkName** parameter does not specify a valid event sink on the component, the operation raises the **InvalidName** exception.

subscribe

The **subscribe** operation associates the subscriber denoted by the **subscriber** parameter with the event source specified by the **publisherName** parameter. If the **publisherName** parameter does not specify a valid event publisher on the component, the operation raises the **InvalidName** exception. The cookie return value can be used to unsubscribe from the source.

unsubscribe

The **unsubscribe** operation disassociates the subscriber associated with **ck** parameter with the event source specified by the **publisherName** parameter. If the **publisherName** parameter does not specify a valid event source on the component, the operation raises the **InvalidName** exception. If the **ck** parameter does not identify a current subscription on the source, the operation raises the **InvalidConnection** exception.

connect_consumer

The **connect_consumer** operation associates the consumer denoted by the **consumer** parameter with the event source specified by the **emitterName** parameter. If the **emitterName** parameter does not specify a valid event emitter on the

component, the operation raises the **InvalidName** exception. If a consumer is already connected to the emitter, the operation raises the **AlreadyConnected** exception. The cookie return value can be used to disconnect from the source.

disconnect_consumer

The **disconnect_consumer** operation disassociates the currently connected consumer from the event source specified by the **emitterName** parameter, returning a reference to the disconnected consumer. If the **emitterName** parameter does not specify a valid event source on the component, the operation raises the **InvalidName** exception. If there is no consumer connected to the emitter, the operation raises the **NoConnection** exception.

5.8 Homes

An IDL specification may include home definitions. A home definition describes an interface for managing instances of a specified component type. The salient characteristics of a home definition are as follows:

- A home definition implicitly defines an equivalent interface, which can be described in terms of IDL as specified in CORBA 2.3a.
- A home definition must specify exactly one component type that it manages. Multiple home definitions may manage the same component type.

  > *This statement applies only to home and component* types. *An actual component instance is managed by exactly one home instance. A component instance can only exist in the context of a home. Component identities are relative to the home to which they belong. Two homes with different definitions may manage components of the same type, but they may not manage the same instances.*

- A home definition may specify a primary key type. Primary keys are values assigned by the application environment that uniquely identify component instances managed by a particular home. Primary key types must be value types derived from **Components::PrimaryKeyBase**. There are more specific constraints placed on primary key types, which are specified in Section 5.8.3.1, “Primary key type constraints.
- The presence of a primary key specification in a home definition causes home’s equivalent interface to contain a set of implicitly defined operations whose signatures are determined by the types of the primary key and the managed component. These operations are specified in Section 5.8.2.2, “Home definitions with primary keys”.
- Home definitions may include any declarations that are legal in normal interface definitions.
- Home definitions support single inheritance from other home definitions, subject to a number of constraints, which are described in Section 5.8.5, “Home inheritance”. The need to inherit home definitions introduces some complexity into the structure

of home equivalent interfaces. The details of home inheritance and the resulting inheritance in equivalent interfaces is described in Section 5.8.5, “Home inheritance”.

5.8.1 Home header

A <*home_header*> describes fundamental characteristics of a home interface, including the following:

- the home type name
- an optional inherited base home type
- the component type managed by the home
- an optional primary key

5.8.1.1 Syntax

The syntax for a home definition is as follows:

<home_dcl> ::= <home_header> <home_body>

<home_header> ::= “home” <identifier> [<home_inheritance_spec>]
“manages” <scoped_name> [<primary_key_spec>]

<home_inheritance_spec> ::= “:” <scoped_name>

<primary_key_spec> ::= “primaryKey” <scoped_name>

<home_body> ::= “{” <home_export>* “}”

<home_export ::= <export>
| <factory_dcl> “;”
| <finder_dcl> “;”

<factory_dcl> ::= “factory” <identifier> “(“ [<init_param_decls>] “)” [
<raises_expr>]

<finder_dcl> ::= “finder” <identifier> “(“ [<init_param_decls>] “)” [
<raises_expr>]

A <*home_header*> consists of the following elements:

- the keyword **home**
- an <*identifier*> that names the home in the enclosing name scope
- an <*inheritance_spec*>, consisting of a colon “**:**” and a <*scoped_name*> that denotes a previously defined home type
- the keyword **manages**
- a <*scoped_name*> that denotes a previously defined component type

- an optional primary key definition, consisting of the keyword **primaryKey** followed by a <*scoped_name*> that denotes a previously defined value type that is derived from the abstract value type **Components::PrimaryKeyBase.** Additional constraints on primary keys are described in Section 5.8.3.1, “Primary key type constraints”.

5.8.2 Equivalent interfaces

Every home definition implicitly defines a set of operations whose names are the same for all homes, but whose signatures are specific to the component type managed by the home and, if present, the primary key type specified by the home.

Because the same operation names are used for these operations on different homes, the implicit operations cannot be inherited. The specification for home equivalent interfaces accommodates this constraint. A home definition results in the definition of three interfaces, called the *explicit* interface, the *implicit* interface, and the *equivalent* interface. The name of the explicit interface has the form ***<home_name>*Explicit**, where ***<home_name>*** is the declared name of the home definition. Similarly, the name of the implicit interface has the form ***<home_name>*Implicit**, and the name of the equivalent interface is simply the name of the home definition, with the form ***<home_name>***. All of the operations defined explicitly on the home (including explicitly-defined factory and finder operations) are represented on the explicit interface. The operations that are implicitly defined by the home definition are exported by the implicit interface. The equivalent interface inherits both the explicit and implicit interfaces, forming the interface presented to programmer using the home.

> *The same names are used for implicit operations in order to provide clients with a simple, uniform view of the basic life cycle operations—creation, finding, and destruction. The signatures differ to make the operations specific to the storage type (and, if present, primary key) associated with the home. These two goals—uniformity and type safety—are admittedly conflicting, and the resulting complexity of equivalent home interfaces reflects this conflict. Note that this complexity manifests itself in generated interfaces and their inheritance relationships; the model seen by the client programmer is relatively simple.*

5.8.2.1 Home definitions with no primary key

Given a home definition of the following form:

```
home <home_name> manages <component_type>
{
    <explicit_operations>
};
```

The resulting explicit, implicit, and equivalent local interfaces have the following forms:

```
interface <home_name>Explicit
: Components::HomeBase
{
    <equivalent_explicit_operations>
};

interface <home_name>Implicit
: Components::KeylessHomeBase
{
    <component_type> create();
};

interface <home_name> :
<home_name>Explicit,
<home_name>Implicit
{};
```

where <*equivalent_explicit_operations*> are the operations defined in the home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 5.8.4, “Explicit operations in home definitions”.

create

This operation creates a new component instance of the type managed by the home.

5.8.2.2 Home definitions with primary keys

Given a home of the following form:

```
home <home_name>
manages <component_type>
primaryKey <key_type>
{
    <explicit_operations>
};
```

The resulting explicit, implicit, and equivalent interfaces have the following forms:

```
interface <home_name>Explicit
: Compoents::HomeBase
{
    <equivalent_explicit_operations>
};

interface <home_name>Implicit
{
    <component_type> create(in <key_type> key)
    raises (Components::DuplicateKeyValue, Components::InvalidKey);

    <component_type> find(in <key_type> key)
    raises (Components::UnknownKeyValue, Components::InvalidKey);

    void destroy(in <key_type> key)
    raises (Components::UnknownKeyValue, Components::InvalidKey);

    <key_type> get_primary_key(in <component_type> comp);

};

interface <home_name>
: <home_name>Explicit ,
<home_name>Implicit
{};
```

where <*equivalent_explicit_operations*> are the operations defined in the home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 5.8.4, “Explicit operations in home definitions.

create

This operation creates a new component associated with the specified primary key value, returning a reference to the component. If the specified key value is already associated with an existing component managed by the storage home, the operation raises an **DuplicateKeyValue** exception. If the key value was not a well-formed, legal value, the operation raises the **InvalidKey** exception.

find

This operation returns a reference to the component identified by the primary key value. If the key value does not identify an existing component managed by the home, an **UnknownKeyValue** exception is raised. If the key value was not a well-formed, legal value, the operation raises the **InvalidKey** exception.

destroy

This operation removes the component identified by the specified key value. Subsequent requests to any of the component’s facets shall raise a **OBJECT_NOT_EXIST** system exception. If the specified key value does not identify

an existing component managed by the home, the operation shall raise an **UnknownKeyValue** exception. If the key value was not a well-formed, legal value, the operation raises the **InvalidKey** exception.

5.8.3 Primary key declarations

Primary key values must uniquely identify component instances within the scope of the home that manages them. Two component instances cannot exist on the same home with the same primary key value.

Different home types that manage the same component type may specify different primary key types. Consequently, a primary key type is not inherently related to the component type, and vice versa. A home definition determines the association between a component type and a primary key type The home implementation is responsible for maintaining the association between specific primary key values and specific component identities.

> *Note that this discussion pertains to component definitions as abstractions. A particular implementation of a component type may be cognizant of, and dependent upon, the primary keys associated with its instances. Such dependencies, however, are not exposed on the surface of the component type. A particular implementation of a component type may be designed to be manageable by different home interfaces with different primary keys, or it may be inextricably bound to a particular home definition. Generally, an implementation of a component type and the implementation of its associated home are inter-dependent, although this is not absolutely necessary.*

5.8.3.1 Primary key type constraints

Primary key and types are subject to the following constraints:

- A primary key type must be a value type derived from **Components::PrimaryKeyBase**.
- A primary key type must be a concrete type with at least one public state member.
- A primary key type may not contain private state members.
- A primary key type may not contain any members whose type is a CORBA interface reference type, including references for interfaces, abstract interfaces, and local interfaces.
- These constraints apply recursively to the types of all of the members, i.e., members which are structs, unions, value types, sequences or arrays may not contain interface reference types. If a the type of a member is a value type or contains a value type, it must meet all of the above constraints.

5.8.3.2 PrimaryKeyBase

The base type for all primary keys is the abstract value type **Components:PrimaryKeyBase**. The definition of **PrimaryKeyBase** is as follows:

```
module Components {
    abstract valuetype PrimaryKeyBase {};
};
```

5.8.4 Explicit operations in home definitions

A home body may include zero or more operation declarations, where the operation may be a *factory* operation, a *finder* operation, or a normal operation or attribute.

5.8.4.1 Factory operations

The syntax of a factory operation is as follows:

<factory_operation> ::= "factory" <identifier> "(" [<init_param_decls>] ")"
[<raises_expr>]

A factor operation declaration consists of the following elements:

- the keyword **factory**
- an identifier that names the operation in the scope of the home definition
- an optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses
- an optional <*raises_expr*> declaring exceptions that may be raised by the operation

A factory operation is denoted by the **factory** keyword. A factory operation has a corresponding equivalent operation on the home's explicit interface. Given a factory declaration of the following form:

```
home <home_name> manages <component_type> {
    factory <factory_operation_name> (<parameters>)
    raises (<exceptions>);
};
```

The equivalent operation on the explicit interface is as follows:

```
<component_type> <factory_operation_name> ( <parameters> )
raises ( <exceptions> );
```

A factory operation is required to support creation semantics, i.e., the reference returned by the operation shall identify a component that did not exist prior to the operation's invocation.

5.8.4.2 Finder operations

The syntax of a finder operation is as follows:

<finder_operation> ::= “finder” <identifier> “(“ [<init_param_decls>] “)”
[<raises_expr>]

A finder operation declaration consists of the following elements:

- the keyword **finder**
- an identifier that names the operation in the scope of the storage home definition
- an optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses
- an optional <*raises_expr*> declaring exceptions that may be raised by the operation

A factory operation is denoted by the **finder** keyword. A finder operation has a corresponding equivalent operation on the home’s explicit interface. Given a factory declaration of the following form:

home *<home_name>* manages *<component_type>* {
finder *<factory_operation_name>* (*<parameters>*)
raises (*<exceptions>*);
};

The equivalent operation on the explicit interface is as follows:

***<component_type>* *<finder_operation_name>* (*<parameters>*)**
raises (*<exceptions>*);

A finder operation is required to support the following semantics. The the reference returned by the operation shall identify a previously-existing component managed by the home. The operation implementation determines which component’s reference to return based on the values of the operation’s parameters.

5.8.4.3 Miscellaneous exports

All of the exports, other than factory and finder operations, that appear in a home definition are duplicated exactly on the home’s explicit interface.

5.8.5 Home inheritance

Given a derived home definition of the following form:

home *<home_name>*
: *<base_home_name>*
manages *<component_type>*
{
<explicit_operations>
};

The resulting explicit interface has the following form:

```
interface <home_name>Explicit
: <base_home_name>Explicit
{
    <equivalent_explicit_operations>
};
```

where <*equivalent_explicit_operations*> are the operations defined in the home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 5.8.4, “Explicit operations in home definitions. The forms of the implicit and equivalent interfaces are identical to the corresponding forms for non-derived storage homes, determined by the presence or absence of a primary key specification.

A home definition with no primary key specification constitutes a pair ***(H, T)*** where ***H*** is the home type and ***T*** is the managed component type. If the home definition includes a primary key specification, it constitutes a triple ***(H, T, K)***, where ***H*** and ***T*** are as previous and ***K*** is the type of the primary key. Given a home definition ***(H’, T’)*** or ***(H’, T’, K)***, where ***K*** is a primary key type specified on ***H’***, such that ***H’*** is derived from ***H***, then ***T’*** must be identical to ***T*** or derived (directly or indirectly) from ***T***.

Given a base home definition with a primary key ***(H, T, K)***, and a derived home definition with no primary key ***(H’, T’)***, such that ***H’*** is derived from ***H***, then the definition of ***H’*** implicitly includes a primary key specification of type ***K***, becoming ***(H’, T’, K)***. The implicit interface for ***H’*** shall have the form specified for an implicit interface of a home with primary key ***K*** and component type ***T’***.

Given a base home definition ***(H, T, K)***, noting that ***K*** may have been explicitly declared in the definition of ***H***, or inherited from a base home type, and a home definition ***(H’, T’, K’)*** such that ***H’*** is derived from ***H***, then ***T’*** must be identical to or derived from ***T*** and ***K’*** must be identical to or derived from ***K***.

Note the following observations regarding these constraints and the structure of inherited equivalent interfaces:

- If a home definition does not specify a primary key directly in its header, but it is derived from a home definition that does specify a primary key, the derived home inherits the association with that primary key type, precisely as if it had explicitly specified that type in its header. This inheritance is transitive. For the purposes of the following discussion, home definitions that inherit a primary key type are considered to have specified that primary key type, even though it did not explicitly appear in the definition header.
- Operations on **HomeBase** are inherited by all home equivalent interfaces. These operations apply equally to homes with and without primary keys.
- Operations on **KeylessHomeBase** are inherited by all homes that do not specify primary keys
- Implicitly-defined operations (i.e., that appear on the implicit interface) are only visible to the equivalent interface for the specific home type that implies their definitions. Implicitly-defined operations on a base home type are not inherited by a

derived home type. Note that the implicit operations for a derived home may be identical in form to the corresponding operations on the base type, but they are defined in a different name scope.

- Explicitly-defined operations (i.e., that appear on the explicit interface) are inherited by derived home types.

5.8.6 Semantics of home operations

Operations in home interfaces fall into two categories:

- Operations that are defined by the component model. Default implementations of these operations must, in some cases, be supplied by the component-enabled ORB product, without requiring user programming or intervention. Implementations of these operations must have predictable, uniform behaviors. Hence, the required semantics for these operations are specified in detail. For convenience, we will refer to these operations as *orthodox* operations.
- Operations that are defined by the user The semantics of these operations are defined by the user-supplied implementation. Few assumptions can be made regarding the behavior of such operations. For convenience, we will refer to these operations as *heterodox* operations.

Orthodox operations include the following:

- Operations defined on **HomeBase** and **KeylessHomeBase**.
- Operations that appear on the implicit interface for any home.

Heterodox operations include the following:

- Operations that appear in the body of the home definition, including factory operations, finder operations, and normal IDL operations and attributes.

5.8.6.1 Orthodox operations

Because of the inheritance structure described in Section 5.8.5, “Home inheritance”, problems relating to polymorphism in orthodox operations are limited. For the purposes of determining key uniqueness and mapping key values to components in orthodox operations, equality of value types (given the constraints on primary key types specified in Section 5.8.3.1, “Primary key type constraints) are defined as follows:

- Only the state of the primary key type specified in the home definition (which is also the actual parameter type in operations using primary keys) shall be used for the purposes of determining equality. If the type of the actual parameter to the operation is more derived that the formal type, the behavior of the underlying implementation of the operation shall be as if the value were truncated to the formal type before comparison. This applies to all value types that may be contained in the closure of the membership graph of the actual parameter value, i.e., if the type of a member of the actual parameter value is a value type, only the state that constitutes the member’s declared type is compared for equality.

- Two values are equal if their types are precisely equivalent and the values of all of their public state members are equal. This applies recursively to members which are value types.
- If the values being compared constitute a graph of values, the two values are equal only if the graphs are isomorphic.
- Union members are equal if both the discriminator values and the values of the union member denoted by the discriminator are precisely equal.
- Members which are sequences or arrays are considered equal if all of their members are precisely equal, where order is significant.

5.8.6.2 Heterodox operations

Polymorphism in heterodox operations is somewhat more problematic, as they are inherited by homes that may specify more-derived component and primary key types. Assume a home definition ***(H, T, K)***, with an explicit factory operation ***f*** that takes a parameter of type ***K***, and a home definition ***(H', T', K')***, such that ***H'*** is derived from ***H***, ***T'*** is derived from ***T***, and ***K'*** is derived from ***K***. The operation ***f*** (whose parameter type is ***K***) is inherited by equivalent interface for ***H'***. It may be the intended behavior of the designer that the actual type of the parameter to invocations of ***f*** on ***H'*** should be ***K'***, exploiting the polymorphism implied by inheritance of ***K*** by ***K'***. Alternatively, it may be the intended behavior of the designed that actual parameter values of either ***K*** or ***K'*** are legitimate, and the implementation of the operation determines what the appropriate semantics of operation are with respect to key equality.

This specification does not attempt to define semantics for polymorphic equality. Instead, we define the behavior of operations on home that depend on primary key values in terms of abstract tests for equality that are provided by the implementation of the heterodox operations.

Implementations of heterodox operations, including implementations of key value comparison for equality, are user-supplied. This specification imposes the following constraints on the tests for equality of value types used as keys in heterodox operations:

- For any two actual key values A and B, the comparison results must be the same for all invocations of all operations on the storage home.
- The comparison behavior must meet the general definition of equivalence, i.e., it must be symmetric, reflexive, and transitive.

5.8.7 HomeBase interface

The definition of the **HomeBase** interface is as follows:

```
module Components {
    interface HomeBase {
        ComponentDef get_component_def();
        void destroy_component ( in ComponentBase comp);
    };
};
```

get_component_def

The **get_component_def** operation returns an object reference that supports the **ComponentDef** interface, describing the component type associated with the home object.

destroy_component

The **destroy_component** operation causes the component denoted by the reference to cease to exist. Subsequent invocations on the reference will cause an **OBJECT_NOT_EXIST** system exception to be raised. If the component denoted by the parameter does not exist in the container associated with target home object, **destroy_component** raises a **BAD_PARAM** system exception.

5.8.8 KeylessHomeBase interface

The definition of the **KeylessHomeBase** interface is as follows:

```
module Components {
    interface KeylessHomeBase {
        ComponentBase create_component();
    };
};
```

create_component

The **create_component** operation creates a new instance of the component type associated with the home object. A home implementation (in particular, an implementation of a home that specifies a primary key) may choose to disable the parameter-less **create_component** operation, in which case it shall raise a **NO_IMPLEMENT** system exception.

5.9 Home Finders

The **HomeFinder** interface is, conceptually, a greatly simplified analog of the **CosLifecycle::FactoryFinder** interface. Clients can use the **HomeFinder** interface to obtain homes for particular component types, of particularly home types, or homes that are bound to specific names in a naming service.

A reference that supports the **HomeFinder** interface may be obtained from the ORB pseudo-object by invoking **CORBA::ORB::resolve_initial_references**, with the parameter value “ComponentHomeFinder”.

The **HomeFinder** interface is defined by the following IDL:

```
module Components {

    exception HomeNotFound { };

    interface HomeFinder {
        HomeBase find_home_by_component_type (
            in CORBA::RepositoryId comprepid)
            raises (HomeNotFound);
        HomeBase find_home_by_home_type (
            in CORBA::RepositoryId homerepid)
            raises (HomeNotFound);
        HomeBase find_home_by_name (
            in string home_name)
            raises (HomeNotFound);
    };
};
```

find_home_by_component_type

The **find_home_by_component_type** operation returns a reference which supports the interface of a home object that manages the component type specified by the **comprepid** parameter. This parameter contains the repository identifier of the component type required. If there are no homes that manage the specified component type currently registered, the operation raises the **HomeNotFound** exception.

> *Little is guaranteed about the home interface returned by this operation. If the definition of the returned home specified a primary key, there is no generic factory operation available on any standard interface (i.e, pre-defined, as opposed to generated type-specific interface) supported by the home. The only generic factory operation that is potentially available is Components::KeylessHomeBase::create_component. The client must first attempt to narrow the HomeBase reference returned by the find_home_by_component_type to KeylessHomeBase. Otherwise, the client must specific out-of-band knowledge regarding the home interface that may be returned, or the client must be sophisticated enough to obtain the InterfaceDef for the home and use the DII to discover and invoke a create operation on a type-specific interface supported by the home.*

find_home_by_home_type

The **find_home_by_home_type** operation returns a reference that supports the interface of type specified by the repository identifier in the **homerepid** parameter. If there are no homes of this type currently registered, the operation raises the **HomeNotFound** exception.

> *The current LifeCycle find_factories operation returns a sequence of factories to the client requiring the client to choose the one which will create the instance. Based on the experience of the submitters, CORBA components defines operations which allows the server to choose the "best" home for the client request based on its knowledge of workload, etc.*

Since the operation returns a reference to **HomeBase**, it must be narrowed to the specific home type before it can be used.

find_home_by_name

The **find_home_by_name** operation returns a home reference bound to the name specified in the home_name parameter. This parameter is expected to contain a name in the format described in the Interoperable Naming Service specification (orbos/98-10-11), section 4.5, “Stringified Names”. The implementation of this operation may be delegated directly to an implementation of CosNaming, but it is not required. The semantics of the implementation are considerably less constrained, being defined as follows:

- The implementation is free to maintain multiple bindings for a given name, and to return any reference bound to the name.

  It is generally expected that implementations that do not choose to use CosNaming will do so for reasons of scalability and flexibility, in order, for example, to provide a home which is logically more “local” to the home finder (and thus, the client).

- The client’s expectations regarding the returned reference, other than that it support the HomeBase interface, are not guaranteed or otherwise mediated by the home. The fact that certain names may be expected to provide certain home types or qualities of implementation are outside of the scope of this interface, and are not addressed by this specification.

  This is no different than any application of naming services in general. Applications that require clients to be more discriminating are free to use the Trader service, or any other similar mechanism that allows query or negotiation to select an appropriate home. This mechanism is intentionally kept simple.

If the specified name does not map onto a home object registered with the finder, the operation raises the **HomeNotFound** exception.

5.10 Component Configuration

The CORBA component model provides mechanisms to support the concept of component *configurability.*

Experience has proven that building re-usable components involves making difficult trade-offs between providing well-defined, reasonably-scoped functionality, and providing enough flexibility and generality to be useful (or re-useful) across a variety of possible applications. Packaging assumptions of the component architecture preclude customizing a component’s behavior by directly altering its implementation or (in most cases) by deriving specialized sub-types. Instead, the model focuses on extension and customization through delegation (e.g., via dependencies expressed with uses declarations) and configuration. Our assumption is that generalized components will typically provide a set of optional behaviors or modalities that can be selected and adjusted for a specific application.

The configuration framework is designed to provide the following capabilities:

- *The ability to define attributes on the component type that are used to establish a component instance's configuration. Component attributes are intended to be used during a component instance's initialization to establish its fundamental behavioral properties. Although the component model does not constrain the visibility or use of attributes defined on the component, it is generally assumed that they will not be of interest to the same clients that will use the component after it is configured. Rather, it is intended for use by component factories or by deployment tools in the process of instantiating an assembly of components.*
- *The ability to define a configuration in an environment other than the deployment environment (e.g., an assembly tool), and store that configuration in a component package or assembly package to be used subsequently in deployment.*
- *The ability to define such a configuration without having to instantiate the component type itself.*
- *The ability to associate a pre-defined configuration with a component factory, such that component instances created by that factory will be initialized with the associated configuration.*
- *Support for visual, interactive configuration tools to define configurations. Specifically, the framework allows component implementors to provide a configuration manager associated with the component implementation. The configuration manager interface provides descriptive information to interactive users, constrains configuration options, and performs validity checks on proposed configurations.*

The CORBA component model allows a distinction to be made between interface features that are used primarily for configuration, and interface features that are used primarily by application clients during normal application operation. This distinction, however, is not precise, and enforcement of the distinction is largely the responsibility of the component implementor.

It is the intent of this specification (and a strong recommendation to component implementors and users) that operational interfaces should be either provided interfaces or a supported interface. Features on the component interface itself, other than provided interfaces, (i.e., receptacles, event sources and sinks) are generally intended to be used for configuration, although there is no structural mechanism for limiting the visibility of the features on a component interface. A mechanism is provided for defining configuration and operational phases in a component's life cycle, and for disabling certain interfaces during each phase.

The distinction between configuration and operational interfaces is often hard to make in practice. For example, we expect that operational clients of a component will want to receive events generated by a component. On the other hand, some applications will want to establish a fixed set of event source and sink connections as part of the overall application structure, and will want to prevent clients from changing those connections. Likewise, the responsibility for configuration may be hard to assign—in some applications the client that creates and configures a component may be the same client that will use it operationally. For this reason, the CORBA component model provides general guidelines and optional mechanisms that may be

employed to characterize configuration operations, but does not attempt to define a strict separation of configuration and operational behaviors.

5.10.1 Exclusive configuration and operational life cycle phases

A component implementation may be designed to implement an explicit configuration phase of its life cycle, enforcing serialization of configuration and functional operation. If this is the case, the component life cycle is divided into two mutually exclusive phases, the *configuration phase* and the *operational phase*.

The **configuration_complete** operation (inherited from **Components::ComponentBase**) is invoked by the agent effecting the configuration to signal the completion of the configuration phase. The **InvalidConfiguration** exception is raised if the state of the component configuration state at the time **configuration_complete** is invoked does not constitute an acceptable configuration state. It is possible that configuration may be a multi-step process, and that the validity of the configuration may not be determined until the configuration process is complete. The **configuration_complete** operation should not return to the caller until either 1) the configuration is deemed invalid, in which case the **InvalidConfiguration** exception is raised, or 2) the component instance has performed whatever work is necessary to consolidate the final configuration and is prepared to accept requests from arbitrary application clients.

In general, component implementations should defer as much consolidation and integration of configuration state as possible until configuration_complete is invoked. In practice, configuring a highly-connected distributed object assembly has proven very difficult, primarily because of subtle ordering dependencies that are difficult to discover and enforce. If possible, a component implementation should not be sensitive to the ordering of operations (interface connections, configuration state changes, etc.) during configuration. This is one of the primary reasons for the definition of configuration_complete.

5.10.1.1 Enforcing exclusion of configuration and operation

The implementation of a component may choose to disable changes to the configuration after **configuration_complete** is invoked, or to disable invocations of operations on provided interfaces until **configuration_complete** is invoked. If an implementation chooses to do either (or both), an attempt to invoke a disabled operations should raise a **BAD_INV_ORDER** system exception.

Alternatively, a component implementation may choose not to distinguish between configuration phase and deployment phase. In this case, invocation of **configuration_complete** will have no effect.

The component implementation framework provides standard mechanisms to support disabling operations during configuration or operation. Certain operations are implemented by the component implementation framework (see Chapter 6), and may not be disabled.

5.11 Configuration with attributes

A component's configuration is established primarily through its attributes. An *attribute configuration* is defined to be a description of a set of invocations on a component's attribute set methods, with specified values as parameters.

There are a variety of possible approaches to attribute configuration at run time, depending on the design of the component implementation and the needs of the application and deployment environments. The CORBA component model defines a set of basic mechanisms to support attribute configuration. These mechanisms can be deployed in a number of ways in a component implementation or application.

5.11.1 Attribute declaration syntax

The CORBA Component specification modifies the existing definition of attributes to add the ability to raise independent exceptions on the attribute's accessor and mutator operations. The modified syntax for attributes is as follows:

<attr_dcl> ::= <readonly_attr_spec>
| <attr_spec>

<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
<readonly_attr_declarator>

<readonly_attr_declarator> ::= <simple_declarator> [<raises_expr>]
| <simple_declarator> { "," <simple_declarator> }*

<attr_dcl> ::= ["readonly"] "attribute" <param_type_spec>
<simple_declarator> { "," <simple_declarator> }*

<attr_spec> ::= "attribute" <param_type_spec> <attr_declarator>

<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
| <simple_declarator> { "," <simple_declarator> }*

<attr_raises_expr> ::= <get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>

<get_excep_expr> ::= "getRaises" <exception_list>

<set_excep_expr> ::= "setRaises" <exception_list>

<exception_list> ::= "(" <scoped_name> { "," <scoped_name> }* ")"

These modifications to the existing attribute declaration syntax allow attribute get and set methods to raise user-defined exceptions. Note the following characteristics of the extended attribute declaration syntax:

- All existing attribute declarations using the previous syntax are still valid, and produce exactly the same results.

- When an attribute declaration raises an exception (on get, set or both), the declaration may not contain multiple declarators.

5.11.2 Language mapping responsibilities

The correspondence between an attribute declaration on an interface and the corresponding operations exposed to a programmer are defined by language mappings; there is no equivalent IDL for operations.

Language mappings shall specify accessor and mutator operations that are capable of raising the exceptions described in the attribute declaration.

5.11.3 Behavior

Component implementations are required to supply the behavior of attribute operations.

> *Attributes are intended to be reflected in the component's internal state, but should not be taken as a concrete description of internal state. The internal state of a component is not visible to the component's clients, and the attribute declarations on a component type definition do not necessarily imply the existence of corresponding concrete state variables in the component. Attribute declarations are syntactic abbreviations for operations to examine and (optionally) set abstract state.*

5.11.4 Attribute Configurators

A configurator is an object that encapsulates a specific attribute configuration that can be reproduced on many instances of a component type. A configurator may invoke any operations on a component that are enabled during its configuration phase. In general, a configurator is intended to invoke attribute set operations on the target component.

5.11.4.1 The Configurator interface

The following interface is supported by all configurators:

```
module Components {

    interface Configurator {
        void configure(in ComponentBase comp)
        raises WrongComponentType;
    };

};
```

configure

The **configure** operation establishes its encapsulated configuration on the target component. If the target component is not of the type expected by the configurator, the configure operation raises the **WrongComponentType** exception.

5.11.4.2 The StandardConfigurator interface

The **StandardConfigurator** has the following definition:

```
module Components {

    valuetype ConfigValue {
        FeatureName name;
        any value;
    };

    typedef sequence<ConfigValue> ConfigValues;

    interface StandardConfigurator : Configurator {
        void set_configuration (in ConfigValues descr);
    };
};
```

The **StandardConfigurator** interface supports the ability to provide the configurator with a set of values defining an attribute configuration.

set_configuration

The **set_configuration** operation accepts a parameter containing a sequence of **ConfigValue** instances, where each **ConfigValue** contains the name of an attribute and a value for that attribute, in the form of an any. The **name** member of the **ConfigValue** type contains the unqualified name of the attribute as declared in the component definition IDL. After a configuration has been provided with **set_configuration**, subsequent invocations of **configure** will establish the configuration on the target component by invoking the set operations on the attributes named in the value set, using the corresponding values provided in the anys. Invocations on attribute set methods will be made in the order in which the values occur in the sequence.

5.11.5 Factory-based configuration

Factory operations on home objects may participate in the configuration process in a variety of ways.

- A factory operation may be explicitly implemented to establish a particular configuration.
- A factory operation may apply a configurator to newly-created component instances. The configurator may be supplied by an agent responsible for deploying a component implementation or a component assembly.
- A factory operation may apply an attribute configuration (in the form of a **Components::ConfigValues** sequence) to newly-created instances. The attribute configuration may be supplied to the home object by an agent responsible for deploying a component implementation or a component assembly.

- A factory operation may be explicitly implemented to invoke **configuration_complete** on newly-created component instances, or to leave component instances open for further configuration by clients.
- A factory operation may be directed by an agent responsible for deploying a component implementation or assembly to invoked **configuration_complete** on newly-created instances, or to leave them open for further configuration by clients.

If no attribute configuration is applied by a factory or by a client, the state established by the component implementation's instance initialization mechanism (e.g., the component servant constructor) constitutes the default configuration.

5.11.5.1 HomeConfiguration interface

The implementation of a component type's home object may optionally support the **HomeConfiguration** interface. The **HomeConfiguration** interface is derived from **Components::HomeBase**. In general, the **HomeConfiguration** interface is intended for use by an agent deploying a component implementation into a container, or an agent deploying an assembly.

The **HomeConfiguration** interface allows the caller to provide a **Configurator** object and/or a set of configuration values that will be applied to instances created by factory operations on the home object. It also allows the caller to cause the home object's factory operations to invoke **configuration_complete** on newly-created instances, or to leave them open for further configuration.

The **HomeConfiguration** allows the caller to disable further use of the **HomeConfiguration** interface on the home object.

> *The Configurator interface and the HomeConfiguration interface are designed to promote greater re-use, by allowing a component implementor to offer a wide range of behavioral variations in a component implementation. As stated previously, the CORBA component specification is intended to enable assembling applications from pre-built, off-the-shelf component implementations. An expected part of the assembly process is the customization (read: configuration) of a component implementation, to select from among available behaviors the behaviors suited to the application being assembled. We anticipate that assemblies will need to define configurations for specific component instances in the assembly, but also that they will need to define configurations for a deployed component type, i.e., all of the instances of a component type managed by a particular home object.*

The **HomeConfiguration** interface is defined by the following IDL:

```
module Components {

    interface HomeConfiguration : HomeBase {
        void set_configurator (in Configurator cfg);
        void set_configuration_values(
            in ConfigValues config);
        void complete_component_configuration(in boolean b);
        void disable_home_configuration();
    };

};
```

set_configurator

This operation establishes a configurator object for the target home object. Factory operations on the home object will apply this configurator to newly-created instances.

set_configuration_values

This operation establishes an attribute configuration for the target home object, as an instance of **Components::ConfigValues**. Factory operations on the home object will apply this configurator to newly-created instances.

complete_component_configuration

This operation determines whether factory operations on the target home object will invoke **configuration_complete** on newly-created instances. If the value of the boolean parameter is true, factory operations will invoke **configuration_complete** on component instances after applying any required configurator or configuration values to the instance. If the parameter is false, **configuration_complete** will not be invoked.

home_configuration_complete

This operation is serves the same function with respect to the home object that the **configuration_complete** operation serves for components. This operation disables further use of operations on the **HomeConfiguration** interface of the target home object. If a client attempts to invoked **HomeConfiguration** operations, the request will raise a **BAD_INV_ORDER** system exception. This operation may also be interpreted by the implementation of the home as demarcation between its own configuration and operational phases, in which case the home implementation may disable operations and attributes on the home interface.

If a home object is supplied with both a configurator and a set of configuration values, the order in which **set_configurator** and **set_configuration_values** are invoked determines the order in which the configurator and configuration values will be applied to component instances. If **set_configurator** is invoked before **set_configuration_values**, the configurator will be applied before the configuration values, and vice-versa.

The component implementation framework defines default implementations of factory operations that are automatically generated. These generated implementations will behave as specified here. Component implementors are free to replace the default factory implementations with customized implementations. If a customized home implementation chooses to support the **HomeConfiguration** interface, then the factory operation implementations must behave as specified, with respect to component configuration.

5.12 CORBAComponent Module

This specification defines a module named **Components**, which contains a set of pre-defined interfaces that support the component model. The **Components** module is defined as follows:

```
module Components {

    interface ComponentDef;

    typedef string FeatureName;
    typedef sequence<FeaureName> NameList;

    valuetype Cookie {
        private sequence<octet> cookieValue;
    };

    exception InvalidName { };
    exception InvalidConnection { };
    exception ExceededConnectionLimit { };
    exception AlreadyConnected { };
    exception NoConnection { };
    exception CookieRequired { };
    exception DuplicateKeyValue { };
    exception UnknownKeyValue { };
    exception BadEventType {
        CORBA::RepositoryId expected_event_type
    };
    exception HomeNotFound { };

    interface Navigation {

        valuetype FacetDescription {
            public RepositoryID InterfaceID;
            public FeatureName Name;
        };

        valuetype Facet : FacetDescription {
            public Object Ref;
        };

        typedef sequence<Facet> Facets;

        typedef sequence<FacetDescription>
            FacetDescriptions;

        Object provide_facet( in FeatureName name )
            raises (InvalidName);

        FacetDescriptions describe_facets();

        Facets provide_all_facets();

        Facets provide_named_facets(in NameList names)
            raises (InvalidName);

        boolean same_component( in Object ref );
```

```
};

valuetype ConnectionDescription {
    public Cookie ck;
    public Object objref;
};

typedef sequence<ConnectionDescription> ConnectedDescriptions;

interface Receptacles {

    Cookie connect (
        in FeatureName name,
        in Object connection )
    raises (
        InvalidName,
        InvalidConnection,
        AlreadyConnected,
        ExceededConnectionLimit);

    void disconnect (
        in FeatureName name,
        in Cookie ck)
    raises (
        InvalidName,
        InvalidConnection,
        CookieRequired,
        NoConnection);

    ConnectionList get_connections (in FeatureName name)
    raises (InvalidName);
};

struct Property {
    PropertyName name;
    PropertyValue value;
};

typedef sequence<Property> EventData;

abstract valuetype EventBase {};

interface EventConsumerBase {
    void push_event(in EventBase evt) raises (BadEventType);
};

interface Events {
    EventConsumerBase
    get_consumer(in FeatureName sinkName)
        raises(InvalidName);
```

```
        Cookie subscribe(in FeatureName publisherName,
            in EventConsumerBase subscriber)
            raises (InvalidName);
        void unsubscribe(in FeatureName publisherName,
            in Cookie ck)
            raises(InvalidName, InvalidConnection);
        void connect_consumer(in FeatureName emitterName,
            in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected);
        EventConsumerBase
        disconnect_consumer(in FeatureName sourceName)
            raises(InvalidName, NoConnection);

};

interface HomeBase {
    ComponentDef get_component_def();
    void destroy_component ( in ComponentBase comp);
};

interface KeylessHomeBase {
    ComponentBase create_component();
};

interface HomeFinder {
    HomeBase find_home_by_component_type (
        in CORBA::RepositoryId comprepid)
        raises (HomeNotFound);
    HomeBase find_home_by_home_type (
        in CORBA::RepositoryId homerepid)
        raises (HomeNotFound);
    HomeBase find_home_by_name (
        in string home_name)
        raises (HomeNotFound);
};

interface Configurator {
    void configure(in ComponentBase comp)
    raises WrongComponentType;
};

valuetype ConfigValue {
    FeatureName name;
    any value;
};

typedef sequence<ConfigValue> ConfigValues;

interface StandardConfigurator : Configurator {
    void set_configuration (in ConfigValues descr);
};
```

```
    interface HomeConfiguration : HomeBase {
        void set_configurator (in Configurator cfg);
        void set_configuration_values(
            in StandardConfigurator::ConfigValues config);
        void complete_component_configuration(in boolean b);
        void disable_home_configuration();
    };

    interface ComponentBase
    : Navigation, Receptacles, Events {
        ComponentDef get_component_def ( );
        HomeBase get_home( );
        void configuration_complete( );
        void destroy();
    };

};
```

5.13 Component Inheritance

The mechanics of component inheritance are defined by the inheritance relationships of the equivalent IDL component interfaces. The following rules apply to component inheritance:

- All interfaces for non-derived component types are derived from **ComponentBase**.
- If a component type directly supports an IDL interface, the component interface is derived from both **ComponentBase** and the supported interface.
- A derived component type may not directly support an interface.
- The interface for a derived component type is derived from the interface of its base component type.
- A component type may have at most one base component type.
- The features of a component that are expressed directly on the component interface are inherited as defined by IDL interface inheritance. These include:
  - operations implied by **provides** statements
  - operations implied by **uses** statements
  - operations implied by **emits** statements
  - operations implied by **consumes** statements
  - attributes

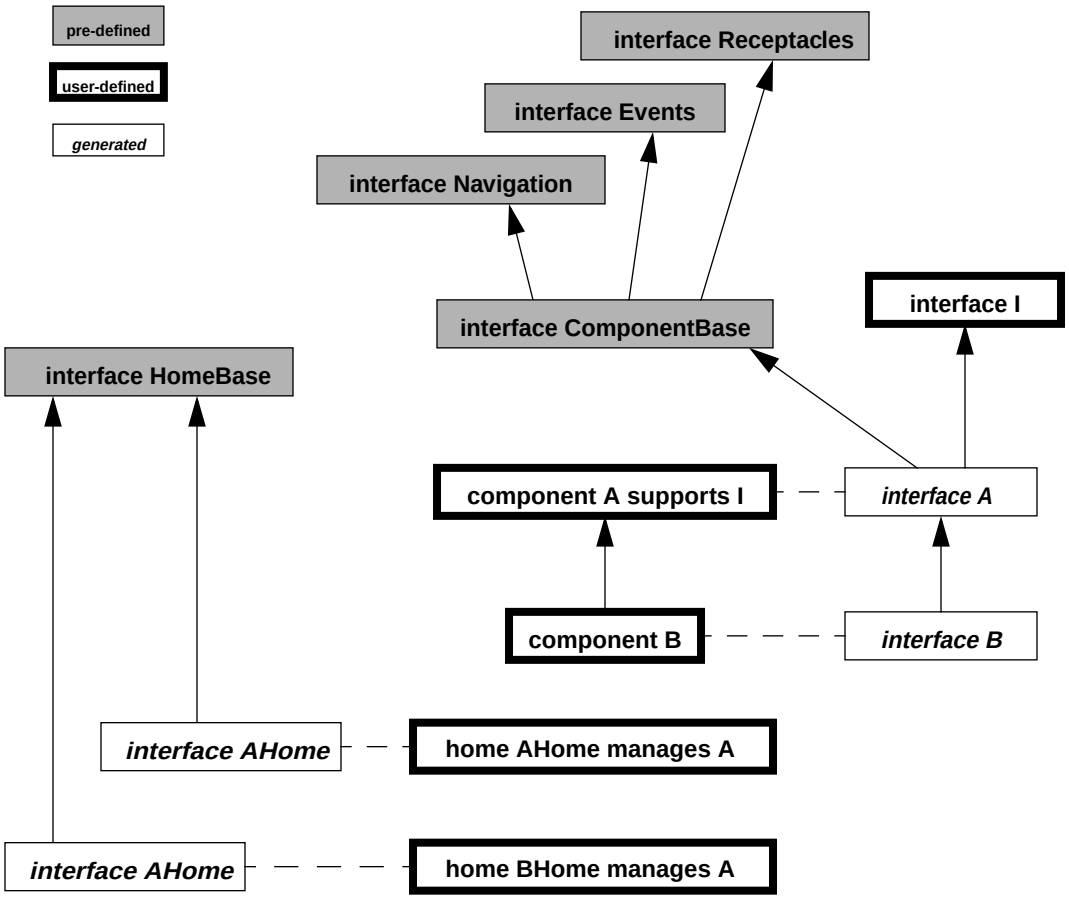

Figure 5-2 Component inheritance and related interface inheritance

5.14 Component Descriptions in the Interface Repository

Component types are described in the Interface Repository by objects that support the **ComponentDef** interface.

```
module CORBA {
    interface ComponentDef : InterfaceDef {
    // details to be supplied in subsequent draft of specification
    };
};
```

ComponentDef is derived from **InterfaceDef**, reflecting the fact that the component meta-type is a specialization and extension of the interface meta-type.

Component Persistence 6

6.1 Persistence and the Component Implementation Framework (CIF)

The Component Implementation Framework (CIF) defines the programming model for constructing component implementations, which includes the programming model for managing persistent state. The elements of the CIF that pertain directly to persistence are described separately in this chapter, and the balance of the CIF is described in <<<>>>. The persistence mechanisms specified in this chapter are best understood when considered in that context.

> *At the time of writing, the OMG process for adopting a Persistent State Specification is in progress, but the outcome (as always) is uncertain. By organizing this document to clearly delineate the specification of state persistence mechanisms, we hope to reduce any future effort required to align the persistence model of the CIF with a putative PSS model. In particular, should a PSS specification be adopted whose specification is sufficiently similar to this one, it may be possible to remove the sections of this document that address persistence, and replace them with a reference to the PSS specification. Otherwise, it will be necessary to either 1) maintain a separate persistence model for components, or 2) define a mapping from the component persistence model to the PSS.*

6.1.1 CIDL, components, and persistence

The focal point of the CIF is Component Implementation Definition Language (CIDL), a declarative language for describing the structure and state of component implementations. Component-enabled ORB products generate implementation skeletons from CIDL definitions. Component builders extend these skeletons to create complete implementations.

CIDL definitions address two distinct aspects of a component implementation—the form of the implementation of the component's behavior (i.e., the organization of the programming artifacts that implement the operations), and the abstract representation of the component's state. Although CIDL as a language addresses both of these issues, only the portion of CIDL that defines persistent state is described in this chapter.

CIDL includes constructs for declaring aggregations of state, called *storage types*, and interfaces for managing instances of those types, called *storage homes* and *persistent stores*. A CIDL implementation definition may optionally associate a storage type with the component implementation, such that the storage type defines the form of the internal state encapsulated by the component. When a component implementation declares an associated storage type in this manner, the CIF and the run-time container environment cooperate to manage the persistence of the component state automatically. The coupling of storage types with component types is described in <<<>>>.

6.2 Component persistence

6.2.1 Persistence concepts

The CIF defines the following concepts pertaining to persistence:

6.2.1.1 Storage type

A storage type is a meta-type defined CIDL. A storage type defines an abstract state that may be specified as the state of an executor (i.e., a component implementation), in which case the component's state will be automatically managed by the generated implementation of the executor, the generated implementation of an associated home executor, and the container. Storage types may also be explicitly managed by a component or object implementation.

6.2.1.2 Storage object

Storage objects are instances of storage types. A storage object is an abstract entity that conceptually exists in the persistent store. A storage object associates an identity (specifically, a Persistent ID or PID) with a state vector. A storage object is created and managed by a persistent store, through the agency of a storage home.

6.2.1.3 Incarnation

An incarnation is a programming artifact that manifests a particular storage object in an execution context. An incarnation provides a programming interface through which the storage object's state can be accessed and modified. An incarnation hides storage implementation details, such as the physical state being managed by the persistent store, actual storage and retrieval of physical data, the implementation of transactional semantics, cache management, and so on.

6.2.1.4 Storage home

A storage home defines an interface for managing instances of a specified storage type. A storage home interface includes operations to create storage objects, find incarnations of storage objects, and destroy instances of the storage type associated with the manager.

6.2.1.5 Persistent store

A persistent store defines a scope within which storage objects are identified and managed. A persistent store is the primary point of contact between the application and the storage mechanisms. A persistent store manages the state of storage objects, and maintains ACID properties of operations on the state (if required). A persistent store provides storage homes through which component implementations can create, incarnate, and destroy storage objects.

6.2.1.6 Persistent ID (or PID)

A PID is a value that uniquely identifies a storage object within the scope of the persistent store to which it belongs. There is a one-to-one mapping between storage objects and PID values. PID values are provided by the persistent store, and are opaque to servant logic and applications.

6.2.1.7 Primary keys

A storage home may optionally define a *primary key*. A primary key declaration specifies a value type, which is derived from an abstract base value type, **Components::PrimaryKeyBase**. Values of the type specified in the storage home definitions are uniquely associated with storage objects managed by that storage home. These values act as *keys*—identities assigned to storage objects by the application at the time of their creation. The storage home is responsible for maintaining the association between a set of key values and their respective storage objects. A primary key declaration implicitly declares a factory operation, a finder operation, and a destructor operation on the storage home. These operations create new component instances with specified key values, map key values onto PIDs or incarnations, and destroy component instances with specified key values, respectively.

6.3 Component Implementation Definition Language (CIDL)

This specification defines Component Implementation Definition Language (CIDL), an adjunct to IDL for describing characteristics of state management and component implementations that can be automatically generated. CIDL definitions include descriptions of a component implementation's persistent state, interfaces for managing instances of that state, and descriptions of the home executor and the component executor.

CIDL is a separate language from IDL, but is closely related to it. The relationship between IDL and CIDL is summarized in the following observations:

- CIDL is used to define the structure of component implementations and the state that they manage. CIDL definitions are not visible beyond the scope of a component's implementation context.
- CIDL can import the contents of specified IDL name scopes from an interface repository or functional equivalent thereof. CIDL constructs can make use of imported IDL types in CIDL definitions.
- CIDL can also be used directly to define a large subset of IDL types, essentially all types but components and interfaces. The intersection of IDL and CIDL grammars is large, as most of the volume of IDL grammar describes constant and data type definitions. Although CIDL data types may be identical to IDL data types, there is no provision for importing CIDL definitions into IDL.
- This specification does not at present define a standard meta-data repository for CIDL constructs, analogous to the interface repository for IDL, though one may be defined in the future if the need arises. However, type identifiers are assigned to CIDL constructs, and are exported through programming interfaces of constructs for availability at run time. The assignment of type identifiers to CIDL constructs is described in <<<>>>.

  The need for a standard meta-data representation is questionable, since CIDL is inherently limited to implementation concerns, primarily with respect to code generation for servants. Unlike the interface repository, a standard CIDL repository is not required for product interoperability.

6.3.1 Grammar description syntax

The syntax used to describe CIDL Grammar is identical to that used to define IDL, and is described in section 3.1 of the CORBA 2.3 specification (ptc/98-12-04).

6.3.2 Lexical conventions

With the exception of keywords, CIDL lexical conventions are identical to those of IDL, and are described in section 3.2 of the CORBA 2.3 specification (ptc/98-12-04).

CIDL defines all of the keywords in IDL as CIDL keywords, whether they appear as terminals in CIDL grammar or not. These are as follows:

abstract	**double**	**long**	**readonly**	**unsigned**
any	**enum**	**module**	**sequence**	**union**
attribute	**exception**	**native**	**short**	**Val ueBase**
boolean	**FALSE**	**Object**	**string**	**valuetype**
case	**fixed**	**octet**	**struct**	**void**
char	**float**	**oneway**	**supports**	**wchar**
const	**in**	**out**	**switch**	**wstring**
context	**init**	**private**	**TRUE**	
custom	**inout**	**public**	**truncatable**	
default	**interface**	**raises**	**typedef**	

Issue – add new component keywords

In addition, CIDL specifies the following keywords:

factory	**local**	**provides**	**storedAs**
finder	**localBase**	**servant**	**strong**
import	**persistentStore**	**storage**	**supports**
incarnationBase	**primaryKey**	**storageHome**	**weak**

6.3.3 CIDL Grammar

As noted above, the intersection of IDL and CIDL is large. To avoid reproducing most of the IDL grammar here, this specification defines a new starting production, and productions for constructs specific to CIDL. All non-terminal symbols that appear in the CIDL grammar below, but which are not defined therein, take their definitions from the IDL grammar specified in the CORBA 2.3 specification (ptc/98-12-04).

Although this section presents the complete CIDL grammar (modulo the intersection with IDL grammar), only the constructs that pertain to persistence are described. The remaining constructs that define form of behavioral units and their relationships to storage types are described in <<<>>>.

The following BNF productions, in conjunction with productions from IDL grammar necessary to resolve non-terminals, define the grammar for CIDL:

```
<cidl_specification> ::= <cidl_imports> <cidl_modules>

<cidl_imports> ::= <cidl_import>*

<cidl_modules> ::= { <cidl_module> “;” }+

<cidl_import> ::= “import” <cidl_imported_scope> “;”

<cidl_imported_scope> ::= <scoped_name> | <string_literal>

<cidl_module> ::= “module” <identifier> “{” <cidl_definition>* “}”

<cidl_definition> ::=  <type_dcl> “;”
                     | <const_dcl> “;”
                     | <except_dcl> “;”
                     | <value> “;”
                     | <cidl_module> “;”
                     | <storage> “;”
                     | <storage_home> “;”
                     | <persistent_store> “;”
                     | <executor> “;”
                     | <home_executor> “;”
                     | <type_id_dcl> “;”

<storage> ::= <storage_dcl>
                     | <storage_forward_dcl>

<storage_forward_dcl> ::= “storage” <identifier>

<storage_dcl> ::= <storage_header> <storage_body>

<storage_header>  ::= “storage” <identifier>
                     [ <storage_inheritance_spec> ]

<storage_inheritance_spec> ::= “:” <scoped_name>

<storage_body> ::= “{” <storage_member>* “}”

<storage_member> ::= <nested_storage_member>
                     | <atomic_member>

<nested_storage_member> = <independent_storage_member>
                     | <dependent_storage_member>

<independent_storage_member> ::= [ “readonly” ] <reference_modifier>
                     <independent_storage_type> <simple_declarator> “;”

<reference_modifier> ::= “strong” | “weak”

<independent_storage_type> ::= <storage_type>
                     | <storage_sequence_spec>
```

```
<storage_type> ::= <storage_dcl>
                 | <storage_type_spec>

<storage_sequence_spec> ::= “sequence” “<” <storage_type_spec> “>”

<storage_type_spec> ::= <scoped_name> | “incarnationBase”

<dependent_storage_member> ::= <storage_type> <simple_declarator> “;”

<atomic_member> ::= [ “readonly” ] <type_spec> <declarators> “;”

<storage_home> ::= <storage_home_header> <storage_home_body>

<storage_home_header> ::= “storageHome” <identifier> [
                 <storage_type_inheritance_spec> ] “manages”
                 <scoped_name> [ <primary_key_spec> ]

<storage_home_inheritance_spec> ::= “:” <scoped_name>

<primary_key_spec> ::= “primaryKey” <scoped_name>

<storage_home_body> ::= “{” <storage_home_member>* “}”

<storage_home_member> ::= <storage_home_operation> “;”

<storage_member_name> ::= <identifier>

<storage_home_operation> ::= <factory_operation>
                 | <finder_operation>
                 | <local_operation>

<factory_operation> ::= “factory” <identifier> “(“ [ <init_param_decls> ] “)” [
                 <raises_expr> ]

<finder_operation> ::= “finder” <identifier> “(“ [ <init_param_decls> ] “)” [
                 <raises_expr> ]

<local_operation> ::= <local>

<persistent_store> ::= <persistent_store_header> <persistent_store_body>

<persistent_store_header> ::= “persistentStore” <identifier>

<persistent_store_body> ::= “{” <persistent_store_member>+ “}”

<persistent_store_member> ::= <storage_home_spec> “;”
                 | <local_operation> “;”

<storage_home_spec> ::= “provides” <storage_home_type_spec>
                 <simple_declarator>
```

<storage_home_type_spec> ::= <scoped_name>

The following productions are included for completeness, but are not discussed in this chapter. The discussion of executor definitions is provided in <<<>>>.

<executor> ::= <executor_header> <executor_body>

<executor_header> ::= <executor_category> <identifier> “implements”
<scoped_name> [<storage_spec>]

<executor_category> ::= “service”
| “session”
| “entity”

<storage_spec> ::= “storedAs” <scoped_name>

<executor_body> ::= “{” <executor_member>+ “}”

<executor_member> ::= “;”

** ::= “segment” <identifier> [<storage_spec>] <segment_body>**

<segment_body> ::= “{” <segment_member>+ “}”

<segment_member> ::= <target_descr>

<target_descr> ::= “provides” <identifier> { “,” <identifier> }* “;”

<home_executor> ::= <home_executor_header> <home_executor_body>

<home_executor_header> ::= <executor_category> “home” <identifier>
“implements” <scoped_name> [“primaryKey”
<scoped_name>] [<delegation_spec>]
[<storage_spec>]

<delegation_spec> ::= “delegatesTo” <scoped_name>

<home_executor_body> ::= “{” <home_executor_member>* “}”

<home_executor_member> ::= <operation_descr> “;”
| <key_descr> “;”

<operation_descr> ::= <identifier> “delegatesTo” < scoped_name>

6.3.4 CIDL type identifiers

CIDL constructs are assigned type identifiers in a manner similar to IDL RepositoryIds. The following rules define the assignment of type identifiers and the form of the identifiers:

- The default format for CIDL type identifiers is identical to IDL format repository identifiers, excepts that the format name prefix is “CIDL:”, rather than “IDL:”.
- By default, all CIDL constructs which are exact analogs of IDL constructs are assigned default type identifiers in CIDL format.
- The following CIDL constructs are assigned default identifiers in CIDL format:

storage storageHome persistentStore executor

- The **typeId** and **typePrefix** declarations defined for IDL behave exactly in CIDL as they do in IDL. These mechanisms are described in <<<>>>.

6.4 CIDL Specification

The grammar for CIDL specifications is described by the following BNF:

<cidl_specification> ::= <cidl_imports> <cidl_modules>

<cidl_imports> ::= <cidl_import>*

<cidl_modules> ::= { <cidl_module> “;” }+

A CIDL specification consists of zero or more imports, followed by one or more module definitions. Unlike IDL, there is no anonymous name scope in CIDL. Declarations (other than module declarations themselves) cannot be made outside of a module scope in CIDL.

6.5 Import

The import mechanism for IDL is described in Section 4.2 on page 24. The same mechanism is available in CIDL, with some minor differences. Specifically, import statements may be used to import name scopes into CIDL specifications from both IDL and CIDL specifications. Although CIDL cannot be used to specify some IDL constructs (e.g., interfaces and components), it can import their definitions into CIDL specifications, where they may be used as constituents of CIDL constructs.

The grammar for a CIDL import statement is isomorphic to that of an IDL import statement:

<cidl_import> ::= “import” <cidl_imported_scope> “;”

<cidl_imported_scope> ::= <scoped_name> | <string_literal>

The <*cidl_imported_scope*> non-terminal may be either a fully-qualified scoped name denoting an IDL or CIDL name scope, or a string containing the interface repository ID of an IDL name scope or CIDL name scope.

As with imports in IDL, a CIDL specification that imports name scopes from IDL and/or CIDL must be interpreted in the context a well-defined set of IDL and/or CIDL specifications that constitute the space from within which name scopes are imported.

Since there is not standard repository for CIDL meta-data, the means by which a particular set of IDL and CIDL specifications is associated with the context in which a CIDL specification is interpreted is not specified

> *At present, we do not specify a repository structure for CIDL. The need for a standard meta-data representation is questionable, since CIDL is inherently limited to implementation concerns, primarily with respect to code generation for servants. Unlike the interface repository, a standard CIDL repository is not required for product interoperability.*

The effects of an import statement in CIDL are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in CIDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL and CIDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- CIDL module definitions may not re-open modules defined in imported IDL name scopes. This rule precludes the possibility of any CIDL declaration having the same fully-qualified name as an imported IDL declaration. CIDL module definitions may, however, re-open modules defined in imported CIDL name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.
- When an name scope is imported, the names of the enclosing scopes in the fully-qualified path name of the enclosing scope are *exposed* within the context of the importing specification, but their contents are not imported. An importing specification may not re-define or re-open a name scope which has been exposed (but not imported) by an import statement.
- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this specification, IDL name scopes that can be imported (i.e., specified in an import statement) into CIDL include the following: **module**, **interface**, **valuetype**, **struct**, **union**, and **exception**. CIDL name scopes that can be imported into other CIDL specifications include the following: **module**, **incarnation**, **storage**, **local**, **struct**, **union**, and **exception**.
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This specification does not define any normative relationship between units specification and units of generation and/or compilation for any language mapping.

6.6 CIDL modules

6.6.1 Syntax

The syntax for a CIDL module definition is as follows:

<cidl_module> ::= “module” <identifier> “{” <cidl_definition>* “}”

<cidl_definition> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <value> “;”
| <cidl_module> “;”
| <storage> “;”
| <storage_home> “;”
| <persistent_store> “;”
| <executor> “;”
| <home_executor> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”

A module defines a name scope, within which the contained definitions are named. Note that CIDL definitions may not exist outside of a module definition; CIDL does not support an anonymous file scope.

6.7 Storage types

Storage types define abstract states that are managed by persistent stores. Storage objects (instances of storage types) are exposed to the application as incarnations, as defined by CIDL language mappings.

A storage declaration consists of a storage header that names the storage type and optionally specifies an inherited base storage type, and a body that declares members of the storage type. An instance of the storage type consists of the aggregation of the states of its members.

6.7.1 Storage Header

6.7.1.1 Syntax

A storage type declaration has the following form:

<storage_header> ::= “storage” <identifier>
[<storage_inheritance_spec>]

<storage_inheritance_spec> ::= “:” <scoped_name>

A storage declaration constitutes a new name scope, nested within the enclosing scope in which the storage is declared.

A storage header consists of the following elements:

- The keyword **storage**.
- An <*identifier*> that names the storage type in the enclosing scope.
- An optional inheritance specification.

A storage forward declaration declares the name of a storage type without defining it. This permits the definition of storage types that refer to each other. The syntax of a forward storage declaration consists of the following elements:

- The keyword **storage**
- An <*identifier*> that names the storage type in the enclosing scope.

The actual definition of the declared storage type must follow later in the specification. Multiple forward declarations of the same storage type name are legal. It is illegal to inherit from a forward-declared storage type whose definition has not yet been seen.

6.7.1.2 Storage type inheritance

A storage type may inherit at most one base storage type. The optional <*scoped_name*> in the storage header identifies a base storage type, from which the storage type being defined is derived. This name must denote a previously defined storage type. Storage type inheritance causes all members defined in the closure of the inheritance tree to be imported into the current storage type's naming scope. Members in derived storage types may not re-define members inherited from base storage types, i.e., member names must be unique within the inheritance hierarchy.

6.7.1.3 Substitutability

When the type of a formal parameter of an operation is a storage type, an incarnation may be passed as the actual parameter if it incarnates the specified formal type or any storage type derived (directly or indirectly) from the formal type. If the formal type of a parameter is **incarnationBase**, an incarnation of any storage type may be passed as the actual parameter.

6.7.1.4 Narrowing

Storage types do not inherently support narrowing operations, as all uses of storage types are local to a single process. Type conversions may be done with the mechanisms provided by programming languages. If a particular language mapping needs to define an explicit narrowing mechanism, it may do so.

6.7.2 Members of storage types

Members of storage types may be *nested storage members* or *atomic members*. The type of a nested storage member is itself a storage type. The type of an atomic storage member is any non-storage type (e.g., IDL primitive types, value types, etc.).

Nested storage members are further sub-divided into two categories—*independent members* and *dependent members*. Independent members are references to distinct storage objects with separate identities and life cycles. Dependent members are contained by the storage types to which they belong. Values assigned to dependent

members have no identities, apart from the identities of the containing storage objects. The life cycle of a dependent member is identical to the life cycle of the storage object that contains it.

All storage types support two kinds of incarnations—*independent* incarnations that have identity, and *dependent* incarnations that have no identity. Independent incarnations may be assigned to independent members of storage types by reference, and used as incarnated states of objects. Dependent incarnations only exist in the context of their enclosing storage objects. The storage type definition describes the form of the storage type, which is invariant across independent and dependent incarnations of the same type.

> *In general, references to independent incarnations should not be type-compatible with representations of dependent incarnations, to discourage inappropriate incarnations from being assigned to the wrong kind of members. When possible, language mappings for object-oriented languages should represent the storage abstraction as an abstract interface that exposes only mutators and accessors for the defined state. The specific types for independent incarnations may mix in an base interface that for independent incarnations that provides operations to manage life cycle and identity. Accessors for dependent members should return the type of the abstract state interface, which cannot be converted to the independent incarnation type.*

6.7.3 Independent storage members

6.7.3.1 Syntax

The syntax of an independent storage member is as follows:

<independent_storage_member> ::= [“readonly”] <reference_modifier>
<independent_storage_type> <simple_declarator> “;”

<reference_modifier> ::= “strong” | “weak”

<independent_storage_type> ::= <storage_type>
| <storage_sequence_spec>

<storage_type> ::= <storage_dcl>
| <storage_type_spec>

An independent member declaration consists of the following elements:

- the optional modifier keyword **readonly**
- a <*reference_modifier*>, either the keyword **strong** or the keyword **weak**
- an <*independent_storage_type*> that defines the type of the member; it must be either of the following forms:
  - a complete storage type definition <*storage_dcl*> nested within the enclosing storage type definition, or

- a <*storage_type_spec*>, either a <*scoped_name*> that must denote a previously declared (but non necessarily defined) storage type, or the keyword **incarnationBase**
- a <*storage_sequence_spec*> that defines the member to be a storage sequence member. Storage sequence members are described in Section 6.7.5, “Storage sequence members.

- a <*simple_declarator*>, which is an identifier that names the member in the scope of the storage type

6.7.3.2 Semantics

An independent member is a reference to a distinct storage object, with its own identity and life cycle. The value of an independent member may be *nil*. Any value assigned to an independent storage member must be a reference to an independent incarnation.

Weak reference

If an independent storage member declaration contained the keyword **weak**, then the life cycle of any storage object whose reference is assigned to the member is unaffected by the life cycle of the owning storage object. Destruction of the owning storage object does not imply destruction of the weak independent member.

Strong reference

If an independent storage member declaration contains the keyword **strong**, then destruction of the owning storage object will cause the destruction of any storage object whose reference is assigned to the strong independent member.

6.7.4 Dependent storage members

6.7.4.1 Syntax

The syntax of a dependent storage member is as follows:

<dependent_storage_member> ::= <storage_type> <simple_declarator> “;”

<storage_type> ::= <storage_dcl> | <storage_type_spec>

<storage_type_spec> ::= <scoped_name> | “incarnationBase”

A dependent member declaration consists of the following elements:

- a <*storage_type*> that defines the type of the member; it must be either of the following forms:
  - a complete storage type definition <*storage_dcl*> nested within the enclosing storage type definition, or

- a <*storage_type_spec*> that must denote a previously declared (but non necessarily defined) storage type or the keyword **incarnationBase**. A dependent member may not be a sequence of a storage type (see Section 6.7.5 on page 104 for discussion of storage member sequences).

- a <*simple_declarator*>, which is an identifier that names the member in the scope of the storage type

This syntax is similar to the independent storage member, with the exception that the **readonly**, **weak**, and **strong** keywords may not appear in the declaration.

6.7.4.2 Semantics

A dependent member is contained by the owning storage object. From the perspective of the storage object abstraction, a dependent member is not held by reference. (Note that a particular language mapping may or may not use the language's notion of reference; this description applies to the conceptual model for dependent members). The value of a dependent member has no distinct identity (i.e., no PID), and its life cycle is identical to that of the owning storage object.

Dependent incarnations cannot be created independently (i.e., by a storage home that manages their storage type). Dependent members are created when their owning storage objects are created. When a storage object with a dependent member is created, the dependent member is assigned a value of its defined type, either a default value, an arbitrary value, or a value determined by the creation operation.

The mutator methods of dependent storage members copy the values of their parameter's members to the corresponding members of the mutator's target. Conceptually, the state of a dependent member is modified by invoking mutators on the dependent incarnation returned by the dependent member's accessor. The behavior of a dependent member's mutator is the equivalent of member-by-member assignment of its own members.

For example:

```
// CIDL

storage duck {
      string s;
long n;
};
storage wabbit {
      weak duck indepDuck;
      duck depDuck;
};
```

The incarnations for A and B would be (roughly):

```
// Java language mapping
interface duckAbstractState {
    string s();
    void s(string s);
    long n();
    void n(long n);
```

```
]

interface duck extends duckAbstractState,
IncarnationBase {}

interface wabbitAbstractState {
    duck indepDuck();
    void indepDuck(duck val);
    duckAbstractState depDuck();
    void depDuck(duckAbstractState val);
}
```

Assume that wabbitHole is a storage home that manages type wabbit. The following example illustrates valid uses of a dependent member:

```
wabbit bugs = wabbitHole.create();
// bugs.indepDuck == nil
// bugs.depDuck != nil, already initializaed with
// dependent incarnation

// the following is preferred for clarity:
bugs.depDuck().s("daffy");

// the following is also legal, but begs the issue
// regarding the ability to get a "reference" for
// the dependent member:
duckAbstractState dk = bugs.depDuck();
dk.n(13);
String str = dk.s();

wabbit peter = wabbitHole.create();

// the following line:
peter.depDuck(bugs.depDuck());

// is the equivalent of:
peter.depDuck().s(bugs.depDuck().s());
peter.depDuck().n(bugs.depDuck().n());

// note also that the java language mapping allows
// independent incarnations as parameters for
// mutators of dependent incarnations:
peter.depDuck(bugs.indepDuck());

// which copies the value of indepDuck, as above

// note that dk cannot be assigned to the
// independent member:
peter.indepDuck(bugs.depDuck());
// illegal! (won't compile)
```

6.7.5 Storage sequence members

Sequences of storage types may only be declared as part of a storage member declaration. Hence, sequences of storage types are always anonymous.

This does not present the same problems that normal anonymous sequences do, since storage sequences cannot be contstructed by applications.

Storage sequences have the following grammar (in the context of independent storage member declarations):

<storage_sequence_spec> ::= “sequence” “<” <storage_type_spec> “>”

Storage sequence members have special semantics, which are summarized as follows:

- The declaration of a sequence member may occur in the context of an independent member declaration.
- The characteristic of independence implied by the declaration, and the nature of the reference stated in the declaration (**strong** or **weak**) apply to the *elements contained in the sequence*, not the sequence itself.
- Storage sequences may only be one-dimensional, i.e., they may not be sequences of sequences of storage types.

  Note that the storage type managed by the sequence may have sequence members, achieving the same effect as sequences of sequences, though in a somewhat more cumbersome manner

- The sequence itself (as opposed to the storage objects in the sequence) is managed by the owning incarnation precisely as though it were a dependent member. Specifically, the sequence has no indentity as a storage object, but it has the characteristic that modifications made through its interface (i.e., the sequence’s interface for assigning storage object references to sequence elements) apply directly to the internal logical state of the incarnation that owns it.
- The mutator for a sequence member does not replace the sequence with another sequence; it replaces the contents of the member sequence with the contents of the argument.
- Members of the sequence are independent members, held by either **strong** or **weak** reference, as specified in the sequence member declaration.

In order to accomodate these semantics, langauge mappings may need to define special representations for sequences for storage objects.

6.7.6 Atomic members

An atomic member is, in general, any type other than an storage type or a sequence of a storage type. An atomic member is atomic from the perspective of the storage type, in that the member value is accessed or mutated as a single, whole entity. Modifications made to an atomic member value obtained from an accessor are not visible to the persistent store until it is assigned with the corresponding mutator. The semantics of atomic members are described in more detail below.

6.7.6.1 Syntax

The syntax for an atomic member is as follows:

<atomic_member> ::= [“readonly”] <type_spec> <declarators> “;”

An atomic member declaration includes the following elements:

- the optional keyword **readonly**
- a <*type_spec*> that describes the type of the member
- one or more <*declarators*> that name the atomic member(s) in the scope of the enclosing storage type definition

The <*type_spec*> may denote any of the following:

- A base IDL or CIDL type (i.e., numeric, character, string, boolean, octet, fixed, any)
- A constructed type (i.e., struct, enum, union) defined in CIDL or imported from IDL
- An object reference type (i.e., an interface type imported from IDL, or Object)
- A value type defined in CIDL or imported from IDL
- A sequence of any of the above types

Storage types, arrays of storage types, and sequences of storage types may not be members of a CIDL struct or union. Storage types and sequences of storage types may not be inserted into values of type any.

If the **readonly** keyword appears in an atomic member declaration, no mutator is provided for the member on incarnations of the storage type.

Although it is legal to define types (with **typedef**) that resolve to arrays of storage types, such an array type may not be a member of a storage type.

6.7.7 Storage object life cycle

Throughout the following discussion, the notion of storage object life cycle pertains to storage objects that are incarnated by independent incarnations. Storage objects incarnated as dependent members do not have distinct life cycles from their owning objects.

The life cycle of a storage object is managed by the persistent store, by way of operations on storage homes. The interface and semantics of storage homes are described in Section 6.8 on page 112.

A storage object is created by a factory operation on a storage home. A storage object must be destroyed by the same storage home that created it. A storage object is said to be *incarnated* when a reference to the storage object’s incarnation exists in an active execution context. The storage object, as an abstraction, exists whether it is incarnated or not. Servant logic can request a storage home to incarnate a storage object with a specified PID value. If the storage home in question specified a primary key, servant logic can request a storage home to incarnate a storage object with a *finder* operation, specifying a key value. User-defined operations on the storage home may also return incarnations.

Note that creation of a storage object does not necessarily imply the immediate incarnation of that storage object. If a storage object does not require state initialization, the creation of a storage object may result only in the existence of a PID, from the perspective of the creating context.

6.7.8 Persistent IDs

A storage object has an identity, called a Persistent ID, or PID, that is unique within the scope of the persistent store associated with the manager that created it. Storage homes (on behalf of their persistent stores) assign PID values to storage objects when they are created. PID values are only meaningful to the persistent stores that allocate them. A storage object's PID value can be obtained from the object's incarnation.

The type of a PID is defined by the following IDL:

```
module Components::Persistence {
    typedef sequence<octet> PersistentId;
};
```

6.7.9 Incarnations

An incarnation is a programming artifact that exposes a storage object's state in an execution context. An incarnation's form in a particular programming language is specified by the CIDL mapping for that language.

The purpose of an incarnation is to de-couple the implementation of an object or component's behavior (i.e., the logic programmed in the servant or executor) from the physical manifestation and management of the object or component's state. A persistent store supplies incarnations that exposes accessor and mutator operations corresponding to the members of the storage type. A servant or executor's implementation manipulates state by invoking these operations on the incarnation. The persistent store that implements the incarnation is free to represent and manage the underlying physical state in any way that satisfies the required semantics.

As described above in Section 6.7.2, "Members of storage types, incarnations may take two forms—*dependent* incarnations and *independent* incarnations. Dependent incarnations manifest storage objects that are contained members of other storage objects. Independent incarnations manifest storage objects that are not members of any other storage object, or are held by reference as members of other storage objects.

A dependent incarnation does not expose any identity for the underlying storage object, since the storage object has no identity apart from its identity as a member of the storage object that contains it. A dependent incarnation provides no operations to manage the life cycle of the underlying contained storage object, since the contained storage object's life cycle is identical to that of the containing storage object. Language mappings for dependent incarnations shall supply the following semantics:

- Each dependent member or read-write independent member declared in the storage type definition shall have corresponding accessor and mutator operations on the dependent incarnation. Read-only independent members shall have only accessor operations.
- The details of managing access to dependent incarnations is determined by each CIDL language mapping. In general, the expected semantics are that references to dependent incarnations cannot be obtained. If these semantics are not supported in the particular programming language (or they would lead to unnecessary clumsiness), then references to dependent incarnations shall not be substitutable for references to independent incarnations. For example, it shall be illegal to obtain the reference to a dependent incarnation of a contained storage object and assign it to independent member of a storage object.

An independent incarnation exposes operations to provide the PID of the storage object it manifests, and to manage the storage object's life cycle. Language mappings shall supply the following semantics for independent incarnations:

- Each dependent member or read-write independent member declared in the storage type definition shall have a corresponding accessor operation and mutator operation on the independent incarnation. Read-only members shall have only an accessor operation.
- Language mappings for independent incarnations shall provide a uniform operation on incarnations for obtaining the storage object's PID.
- Language mappings for independent incarnations shall provide a uniform operation on incarnations for obtaining their associated persistent store.
- Language mappings for independent incarnations shall provide a uniform operation on incarnations for obtaining the associated storage home.
- Language mappings for independent incarnations shall be managed by reference. Any incarnation reference may have the value nil, denoting the absence of an incarnation.
- Language mappings for independent incarnations shall provide a uniform operation on incarnations for destroying the associated storage object.
- Language mappings shall provide a mechanism for obtaining the type identity of a storage type in the abstract (e.g., a static method on the incarnation type, or a helper class). CIDL type identities are described in <<<>>>.

Whether an incarnation is dependent or independent is not a function of the storage type it incarnates. Independent incarnations may be obtained from factory and finder operations on storage homes. Dependent incarnations cannot be explicitly created; they are created as a side effect of the creation of their owning storage objects. If an incarnation is obtained from the accessor for a dependent member of another incarnation, it is a dependent incarnation. A single storage type may simultaneously have some incarnations that are dependent and others that are independent.

Where possible, language mappings for operations on dependent incarnations that manage the abstract state of the underlying storage type shall be identical to the mappings for the same operations on independent incarnations.

6.7.9.1 IncarnationBase type

The keyword **incarnationBase** denotes the generalization of incarnation types. In practice, it acts as an abstract base type for all incarnation types. It plays the same role with respect to storage types that **ValueBase** plays with respect to value types. It can be used in declarations to describe signatures that can accept or return an incarnation of any storage type.

The relevant sections of each language mapping contain the specifics of how the **IncarnationBase** type is mapped to a particular language.

6.7.10 Persistence Semantics

The behavior of incarnations is specified in terms of the behaviors of member accessor and mutator functions. No assumptions can be made regarding the physical state of a given storage object, either as physical state in the active execution context, or physical state stored on some secondary medium. An incarnation presents a *logical* view of a storage object's state, and may employ any desired technique (such as caching, interposition, pre-fetching, etc.) to manage underlying physical state so as to present the correct logical behavior. The precise definition of the logical value of a storage object is determined by the implementation of the persistence mechanism and the context in which the access to the state occurs (e.g., the isolation policies of an underlying transaction mechanism).

6.7.10.1 Creation

Storage objects are created by factory operations on storage homes. A factory operation may establish initial values for any or all of the members of the storage object. The initial values may be derived from the parameters of the factory operation (if any), they may be default values determined by the persistent store or the implementation of the storage home, or they may be undefined.

When a storage object is created, a PID is allocated by the persistent store and assigned to the storage object. The PID is unique to the storage object within the scope of the persistent store, and invariant over the life time of the storage object.

6.7.10.2 Incarnation

The creation of a storage object by a factory operation may or may not result in the incarnation of the storage object, depending on the form of the factory operation. If the factory operation returns a PID, it can be subsequently used to incarnate the storage object. If the factory operation returns an incarnation, the PID can be obtained directly from the incarnation.

6.7.10.3 State access and modification

The storage object's logical state is exposed to an execution context through accessor and mutator functions on an incarnation. An accessor function shall return the current logical value of the member of the storage object. A mutator function shall modify the

logical state of the storage object to reflect the value of the parameter. The relationship between accessor and mutator invocations and management of underlying physical state is not specified. It is the responsibility of the persistence mechanism and other cooperating mechanisms (such as a transaction mechanism) to ensure proper and consistent logical behavior. Different implementations of a persistent store may have different definitions of *proper and consistent logical behavior,* and may define different circumstances under which such behavior can be guaranteed.

6.7.10.4 Incarnation release

The manner in which an incarnation is destroyed or released is determined by the specific programming language mapping. The life cycle of an incarnation is independent of the life cycle of its storage object; releasing or destroying an incarnation does not imply the destruction of the underlying storage object. A particular storage object (as identified by a single PID value) may have multiple incarnations over a period of time, possibly in multiple processes. The persistent store is responsible for maintaining the object's state between (and during) incarnations, within the behavioral limits defined by the particular persistent store.

6.7.10.5 Destruction

A storage object may be destroyed by operations on the storage home, or the destruction operation on the incarnation. After a storage object is destroyed, any attempt to incarnate it with its PID shall fail (the precise manner of failure is specified in the interface definition for storage home). When the storage object associated with an incarnation is destroyed and the incarnation remains, any attempted invocation of an operation on the incarnation shall fail, raising the **OBJECT_NOT_EXIST** system exception.

6.7.10.6 Independent member semantics

For the purposes of this discussion, a storage object that has another storage object for an independent member is called the *parent* storage object; the member is called the *child* storage object. The relationship between a parent storage object and a child object is described by the following observations:

- Parent and child objects have distinct identities (i.e., distinct PIDs).
- The parent object holds a reference for the child object. The value of the reference may be nil, indicating the absence of a child storage object.
- Graphs of storage objects formed by parent-child relationships may be re-entrant or cyclic (e.g., a child may have multiple parents, cyclic relationships may exist). Persistence mechanisms must respect and reproduce these relationships when storage objects are incarnated.
- A child storage object must belong to the same persistent store as its parent(s). All of the storage objects in a parent-child graph must belong to the same persistent store.

- If the keyword **strong** appears in the declaration of a storage member, then the destruction of the parent object will cause the destruction of any child object assigned to the member. If the keyword **weak** appears, the destruction of the parent will not result in the destruction of the child.
- Storage types may have independent members of their own type, or that contain members of their own type, recursively.

The following is allowed:

```
storage duck {
    strong duck next;
};
```

also:

```
storage wabbit; // forward reference
storage duck {
    strong wabbit bugs;
};
storage wabbit {
    strong duck daffy;
    // the following is legal because the independent member
    // bugs of duck prevents unconditional infinite recursion
    duck donald;
};
```

6.7.10.7 Dependent member semantics

Dependent members behave as if their members were direct members of their owning storage objects. If storage type ***A*** had dependent member ***ma*** of type ***B***, and storage type ***B*** had a member ***mb*** (of any type), the semantics of the dependent member ***ma*** would be as though its member ***mb*** were directly a member of ***A***. The intervening dependent member ***ma*** provides the following utility:

- a dependent member aggregates its members under a single name, as a subset of the state of the owning storage object
- a dependent member allows the re-use of the type definition as an aggregation
- a dependent member allows its value to be assigned in a single operation

Storage types may not define dependent members of their own type, directly or indirectly.

Actually, this is prevented by the inability to declare a dependent member of a type which has not been previously defined.

6.7.10.8 Atomic member semantics

The details of assignment and management of atomic members (i.e., members which are non-storage data types) are determined by language mappings, as are the rules for memory management in languages where explicit memory management is an issue. In general, the semantics of atomic member assignment and management shall be value

semantics when efficiency concerns permit it. It is preferred that atomic member values are copied when returned from accessor functions and passed to mutator functions on incarnations.

A language mapping may choose to assign and manage certain types of atomic members by reference. For types whose instances may be physically large, such as value types, sequences, structs, etc., copying the value on every access or mutation may be unreasonably inefficient. Language mappings must clearly specify the assumptions that the servant logic is and is not allowed to make regarding ownership of a datum whose reference is obtained from, or assigned to, an atomic member accessor or mutator.

6.7.10.9 Valuetype atomic members

Sharing semantics

Sharing of value type instances is only defined and maintained with respect to a single atomic member. If a graph of values is constructed such that two separate atomic members (whose types are a value type) share values, the graphs that are reproduced when the storage object is subsequently incarnated will not share corresponding values. The graph reproduced for a specific member will be isomorphic to the graph that was visible from that member when the storage object's previous incarnation was synchronized with storage.

Polymorphism and truncation

Persistent stores are not required to manage value type instances that are more derived than the declared types of the atomic members of the storage types in question. A persistent store implementation may choose to attempt to truncate value type instances to their declared types when they are assigned to members via mutator operations. This applies not only to the value that is assigned directly to the storage member, but also to all values in a graph that are reachable from the root value assigned to the storage member. A persistent store may attempt to truncate each value to the declared type of the reference (i.e., the declared type of the value member) to which it is assigned.

6.8 Storage home

A storage home definition associates a storage type with an interface that supports life cycle management of instances of the storage type. This interface is called the *storage home interface*.

6.8.1 Syntax

The syntax for declaring a storage home is as follows:

<storage_home> ::= <storage_home_header> <storage_home_body>

<storage_home_header> ::= “storageHome” <identifier> [
<storage_type_inheritance_spec>] “manages”
<scoped_name> [<primary_key_spec>]

<storage_home_inheritance_spec> ::= “:” <scoped_name>

<primary_key_spec> ::= “primaryKey” <scoped_name>

<storage_home_body> ::= “{” <storage_home_member>* “}”

<storage_home_member> ::= <storage_home_operation> “;”

<storage_member_name> ::= <identifier>

<storage_home_operation> ::= <factory_operation>
| <finder_operation>
| <local_operation>

A storage home declaration consists of the following elements:

- the keyword **storageHome**
- an <*identifier*> that names the storage home in the enclosing scope
- an optional <*storage_home_inheritance_spec*> consisting of a colon “:” followed by a <*scoped_name*> that denotes a previously defined storage home type
- the keyword **manages**
- a <*scoped name*> that denotes a previously-defined storage type
- an <*primary_key_spec*> consisting of the keyword **primaryKey**, followed by a <*scoped_name*> denoting a primary key type
- a storage home body

A storage home declaration constitutes a name scope.

If a storage home declaration includes an inheritance specification, the storage type managed by the home must be equivalent to, or derived from, the storage type managed by the base storage home type identified in the inheritance specification.

The type denoted by the <*scoped_name*> in the primary key specification must be a value type derived from the base type **Components::PrimaryKeyBase**. Constraints on primary key types are discussed in Section 6.8.4, “Primary key type constraints.

6.8.2 Equivalent local interfaces

The language mappings of storage home interfaces are defined indirectly, in terms of *equivalent local interfaces*. A storage home definition maps deterministically onto a **local** interface, which is called the storage home’s *equivalent local interface*. Local interfaces are defined in Section 4.1 on page 19.

6.8.2.1 Implicit operations and equivalent local interface structure

Every storage home definition implicitly defines a set of operations whose names are invariant across storage homes, but whose signatures are specific to the storage type managed by the storage home and, if present, the primary key type specified by the storage home.

Because the same operation names are used for different storage homes, the implicit operations cannot be inherited. The specification for storage home equivalent local interfaces accommodates this constraint. A storage home definition results in the definition of three interfaces, called the explicit interface, the implicit interface, and the equivalent interface. The name of the explicit interface has the form ***<storage_home_name>*Explicit**, where ***<storage_home_name>*** is the declared name of the storage home definition. Similarly, the name of the implicit interface has the form ***<storage_home_name>*Implicit**, and the name of the equivalent interface is simply the name of the storage home definition, with the form ***<storage_home_name>***. All of the operations defined explicitly on the storage home (including explicitly-defined factory and finder operations) are represented on the explicit interface. The operations that are implicitly defined by the storage home definition are exported by the implicit interface. The equivalent interface inherits both the explicit and implicit interfaces, forming the interface presented to programmer using the storage home.

> *The same names are used for implicit operations in order to provide clients with a simple, uniform view of the basic life cycle operations—creation, finding, and destruction. The signatures differ to make the operations specific to the storage type (and, if present, primary key) associated with the home. These two goals—uniformity and type safety—are admittedly conflicting, and the resulting complexity of equivalent home interfaces reflects this conflict. Note that this complexity manifests itself in generated interfaces and their inheritance relationships; the model seen by the client programmer is relatively simple.*

6.8.2.2 Storage home definitions with no primary key

Given a storage home of the following form:

```
storageHome <storage_home_name> manages <storage_type>
{
    <explicit_operations>
};
```

The resulting explicit, implicit, and equivalent local interfaces have the following forms:

```
local <storage_home_name>Explicit
: Components::Persistence::StorageHomeBase
{
    <equivalent_explicit_operations>
};

local <storage_home_name>Implicit
: Components::Persistence::KeylessHomeBase
{
    Components::Persistence::PersistentId create();

    <storage_type> find_by_pid(
        in Components::Persistence::PersistentId pid
    )
    raises (Components::Persistence::DoesNotExist);
};

local <storage_home_name> :
<storage_home_name>Explicit ,
<storage_home_name>Implicit
{};
```

where <*equivalent_explicit_operations*> are the operations defined in the storage home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 6.8.5, “Explicit operations in storage home definitions.

create

This operation creates a new storage object, returning its PID. The PID can be subsequently to incarnate the storage object with **find_by_pid** on the implicit interface, or with **StorageHomeBase::find_incarnation_by_pid**.

find_by_pid

This operation incarnates the storage object identified by the pid parameter, returning the incarnation. If the pid parameter does not identify a storage object managed by the target storage home, the **DoesNotExist** exception is raised.

6.8.2.3 Storage home definitions with primary keys

Given a storage home of the following form:

```
storageHome <storage_home_name>
manages <storage_type>
primaryKey <key_type>
{
    <explicit_operations>
};
```

The resulting explicit, implicit, and equivalent local interfaces have the following forms:

```
local <storage_home_name>Explicit
: Components::Persistence::StorageHomeBase
{
    <equivalent_explicit_operations>
};

local <storage_home_name>Implicit
{
    <storage_type> create_with_key(in <key_type> key)
    raises (Components::Persistence::AlreadyExists);

    Components::Persistence::PersistentId
    create_pid_with_key(in <key_type> key)
    raises (Components::Persistence::AlreadyExists);

    <storage_name> find_by_key(in <key_type> key)
    raises (Components::Persistence::DoesNotExist);

    <storage_name>
    find_by_pid(in Components::Persistence::PersistentID pid)
    raises (Components::Persistence::DoesNotExist);

    Components::Persistence::PersistentID
    find_pid_by_key(in <key_type> key)
    raises (Components::Persistence::DoesNotExist);

    void remove_by_key(in <key_type> key)
    raises (Components::Persistence::DoesNotExist);

    <key_type> find_key_by_pid(
        in Components::Persistence::PersistentID pid
    )
    raises (Components::Persistence::DoesNotExist);

};

local <storage_home_name>
: <storage_home_name>Explicit ,
<storage_home_name>Implicit
{};
```

where <*equivalent_explicit_operations*> are the operations defined in the storage home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 6.8.5, “Explicit operations in storage home definitions.

create_with_key

This operation creates a new storage object associated with the specified primary key value, and allocates a new PID value for the storage object. An incarnation of the new object is returned. If the specified key value is already associated with an existing storage object managed by the storage home, the operation raises an **AlreadyExists** exception.

create_pid_with_key

This operation creates a new storage object associated with the specified primary key value, and allocates a new PID value for the storage object. The operation returns the new PID value. If the specified key value is already associated with an existing storage object managed by the storage home, the operation raises an **AlreadyExists** exception.

find_by_key

This operation produces an incarnation of the storage object identified by the primary key value. If the key value does not identify an existing storage object managed by the storage home, a **DoesNotExist** exception is raised.

find_by_pid

This operation produces an incarnation for the storage object identified by the specified PID value. If the PID value does not identify an existing storage object managed by the storage home, the operation raises a **DoesNotExist** exception.

find_pid_by_key

This operation returns the PID of the storage object identified by the specified key value. If the key value does not identify an existing storage object managed by the storage home, a **DoesNotExist** exception is raised.

remove_by_key

This operation removes the storage object identified by the specified key value. Subsequent attempts to incarnate the object shall raise a DoesNotExist exception. If the specified key value identify an existing storage object managed by the storage home, the operation shall raise a **DoesNotExist** exception.

find_key_by_pid

This operation returns the primary key value associated with the storage object identified by the specified PID value. If the PID value does not identify an existing storage object managed by the storage home, a **DoesNotExist** exception is raised.

6.8.3 Initial values of created storage objects

When a new storage object is created by any of the implicitly-defined create methods, the initial values of the storage object members shall meet the following constraints:

- All independent storage members shall be nil.

- All members that are value types shall be nil.
- All members that are objects reference types shall return nil object references.
- All sequence members shall be initialized to length zero.
- All string members will be initialized to zero-length strings.
- All anys will be empty.
- The initial values of base data types shall be undefined.

When a new storage object is created by an explicitly-defined factory operation on a storage home, the initial values are defined by the implementation of the operation.

6.8.4 Primary key type constraints

The persistent store is responsible for maintaining the association between primary key values and their respective storage objects, and for being able to map putative values to storage objects.

Primary key and types are subject to the following constraints:

- A primary key type must be a value type derived from **PrimaryKeyBase**.
- A primary key type must be a concrete type with at least one public state member.
- A primary key type may not contain private state members.
- A primary key type may not contain any members whose type is a CORBA interface reference type, including references for interfaces, abstract interfaces, and local interfaces.
- These constraints apply recursively to the types of all of the members, i.e., members which are structs, unions, value types, sequences or arrays may not contain interface reference types. If a the type of a member is a value type or contains a value type, it must meet all of the above constraints.

6.8.5 Explicit operations in storage home definitions

A storage home body may include zero or more operation declarations, where the operation may be a *factory* operation, a *finder* operation, or a *local* operation.

6.8.5.1 Factory operations

The syntax of a factory operation is as follows:

<factory_operation> ::= “factory” <identifier> “(“ [<init_param_decls>] “)”
[<raises_expr>]

A factor operation declaration consists of the following elements:

- the keyword **factory**
- an identifier that names the operation in the scope of the storage home definition

- an optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses
- an optional <*raises_expr*> declaring exceptions that may be raised by the operation

A factory operation is denoted by the **factory** keyword. A factory operation has a corresponding equivalent operation on the storage home's explicit interface. Given a factory declaration of the following form:

factory *<factory_operation_name>* (*<parameters>*) raises (*<exceptions>*);

The equivalent operation on the explicit interface is as follows:

Components::Persistence::PersistentId *<factory_operation_name>* (
**   *<parameters>***
) raises (*<exceptions>*);

A factory operation is required to support creation semantics, i.e., the PID returned by the operation shall identify a storage object that did not exist prior to the operation's invocation. The responsibility for implementing explicitly-defined factory operations is described in Section 6.8.7, "Implementation responsibility.

6.8.5.2 Finder operations

The syntax of a finder operation is as follows:

<finder_operation> ::= "finder" <identifier> "(" [<init_param_decls>] ")"
[<raises_expr>]

A finder operation declaration consists of the following elements:

- the keyword **finder**
- an identifier that names the operation in the scope of the storage home definition
- an optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses
- an optional <*raises_expr*> declaring exceptions that may be raised by the operation

A factory operation is denoted by the **finder** keyword. A finder operation has a corresponding equivalent operation on the storage home's explicit interface. Given a factory declaration of the following form:

finder *<finder_operation_name>* (*<parameters>*) raises (*<exceptions>*);

The equivalent operation on the explicit interface is as follows:

Components::Persistence::PersistentId
***<finder_operation_name>* (*<parameters>*) raises (*<exceptions>*);**

A finder operation is required to support the following semantics. The the PID returned by the operation shall identify a previously-existing storage object managed by the storage home. The operation implementation determines which storage object's PID to

return based on the values of the operation's parameters. The responsibility for implementing explicitly-defined finder operations is described in Section 6.8.7, "Implementation responsibility.

> *Note that the signatures of factory and finder operations are not affected by the storage type associated with the storage home, as one might expect. The operations return a PID value because the most common use of these operations is to provide a PID value that is incorporated into an object reference. Incarnation occurs when the object is activated by a request, most likely with StorageHomeBase::incarnate_by_pid.*

6.8.5.3 Local operations

Local operation syntax is specified and described in Section 4.1 on page 19. The responsibility for implementing explicitly-defined local operations on storage homes is described in Section 6.8.7, "Implementation responsibility.

6.8.6 Storage home inheritance

Given a derived storage home definition of the following form:

```
storageHome <storage_home_name>
: <base_storage_home_name>
manages <storage_type>
{
    <explicit_operations>
};
```

The resulting explicit local interface has the following form:

```
local <storage_home_name>Explicit
: <base_storage_home_name>Explicit
{
    <equivalent_explicit_operations>
};
```

where <*equivalent_explicit_operations*> are the operations defined in the storage home declaration (<*explicit_operations*>), with factory and finder operations transformed to their equivalent operations, as described in Section 6.8.5, "Explicit operations in storage home definitions. The forms of the implicit and equivalent interfaces are identical to the corresponding forms for non-derived storage homes, determined by the presence or absence of a primary key specification.

A storage home definition with no primary key specification constitutes a pair ***(H, T)*** where ***H*** is the storage home type and ***T*** is the managed storage type. If the storage home definition includes a primary key specification, it constitutes a triple ***(H, T, K)***, where ***H*** and ***T*** are as previous and ***K*** is the type of the primary key. Given a storage home definition ***(H', T')*** or ***(H', T', K)***, where ***K*** is a primary key type specified on ***H'***, such that ***H'*** is derived from ***H***, then ***T'*** must be identical to ***T*** or derived (directly or indirectly) from ***T***.

Given a base storage home definition with a primary key ***(H, T, K)***, and a derived storage home definition with no primary key ***(H', T')***, such that ***H'*** is derived from ***H***, then the definition of ***H'*** implicitly includes a primary key specification of type ***K***, becoming ***(H', T', K)***. The implicit interface for ***H'*** shall have the form specified for an implicit interface of a storage home with primary key ***K*** and storage type ***T'***.

Given a base storage home definition ***(H, T, K)***, noting that ***K*** may have been explicitly declared in the definition of ***H***, or inherited from a base storage home type, and a storage home definition ***(H', T', K')*** such that ***H'*** is derived from ***H***, then ***T'*** must be identical to or derived from ***T*** and ***K'*** must be identical to or derived from ***K***.

Note the following observations regarding these constraints and the structure of inherited equivalent interfaces:

- If a storage home definition does not specify a primary key directly in its header, but it is derived from a storage home definition that does specify a primary key, the derived storage home inherits the association with that primary key type, precisely as if it had explicitly specified that type in its header. This inheritance is transitive. For the purposes of the following discussion, storage home definitions that inherit a primary key type are considered to have specified that primary key type, even though it did not explicitly appear in the definition header.
- Operations on **StorageHomeBase** are inherited by all storage home equivalent interfaces. these operations apply equally to homes with and without primary keys.
- Operations on **KeylessStorageHomeBase** are inherited by all storage homes that do not specify primary keys
- Implicitly-defined operations (i.e., that appear on the implicit interface) are only visible to the equivalent interface for the specific storage home type that implies their definitions. Implicitly-defined operations on a base storage type are not inherited by a derived storage type. Note that the implicit operations for a derived storage home may be identical in form to the corresponding operations on the base type, but they are defined in a different name scope.
- Explicitly-defined operations (i.e., that appear on the explicit interface) are inherited by derived storage home types.

6.8.7 Implementation responsibility

Responsibility for the implementations of operations in storage home interfaces fall into two categories:

- Operations whose implementations must be provided by the storage product, without requiring user programming or intervention. Implementations of these operations must have predictable, uniform behaviors between storage products. Hence, the required semantics for these operations are specified in detail. For convenience, we will refer to these operations as *orthodox* operations.
- Operations whose implementations must be specified by the user, either through programming or with unspecified tools supplied with the storage product. The semantics of these operations are defined by the user-supplied implementation. For convenience, we will refer to these operations as *heterodox* operations.

Orthodox operations include the following:

- Operations defined on **StorageHomeBase** and **KeylessStorageHomeBase**.
- Operations that appear on the implicit interface for any storage home.

Heterodox operations include the following:

- Operations that appear in the body of the storage home definition, including factory operations, finder operations, and local operations.

6.8.7.1 Orthodox operations

Because of the inheritance structure described in Section 6.8.6 on page 121, problems relating to polymorphism in orthodox operations are limited. For the purposes of determining key uniqueness and mapping key values to storage objects in orthodox operations, equality of value types (given the constraints on primary key types specified in Section 6.8.4, “Primary key type constraints) are defined as follows:

- Only the state of the primary key type specified in the storage home definition (which is also the actual parameter type in operations using primary keys) shall be used for the purposes of determining equality. If the type of the actual parameter to the operation is more derived that the formal type, the behavior of the underlying implementation of the operation shall be as if the value were truncated to the formal type before comparison. This applies to all value types that may be contained in the closure of the membership graph of the actual parameter value, i.e., if the type of a member of the actual parameter value is a value type, only the state that constitutes the member’s declared type is compared for equality.
- Two values are equal if their types are precisely equivalent and the values of all of their public state members are equal. This applies recursively to members which are value types.
- If the values being compared constitute a graph of values, the two values are equal only if the graphs are isomorphic.
- Union members are equal if both the discriminator values and the values of the union member denoted by the discriminator are precisely equal.
- Members which are sequences or arrays are considered equal if all of their members are precisely equal, where order is significant.

6.8.7.2 Heterodox operations

Polymorphism in heterodox operations is somewhat more problematic, as they are inherited by storage homes that may specify more-derived storage and primary key types. Assume a storage home definition ***(H, T, K)***, with an explicit factory operation ***f*** that takes a parameter of type ***K***, and a storage home definition ***(H’, T’, K’)***, such that ***H’*** is derived from ***H***, ***T’*** is derived from ***T***, and ***K’*** is derived from ***K***. The operation ***f*** (whose parameter type is ***K***) is inherited by equivalent interface for ***H’***. It may be the intended behavior of the designer that the actual type of the parameter to invocations of ***f*** on ***H’*** should be ***K’***, exploiting the polymorphism implied by inheritance of ***K*** by ***K’***. Alternatively, it may be the intended behavior of the designed that actual parameter

values of either ***K*** or ***K'*** are legitimate, and the implementation of the operation determines what the appropriate semantics of operation are with respect to key equality.

This specification does not attempt to define semantics for polymorphic equality. Instead, we define the behavior of operations on storage home that depend on primary key values in terms of abstract tests for equality that are provided by the implementation of the heterodox operations.

Implementations of heterodox operations, including implementations of key value comparison for equality, are user-supplied. This specification imposes the following constraints on the tests for equality of value types used as keys in heterodox operations:

- For any two actual key values A and B, the comparison results must be the same for all invocations of all operations on the storage home.
- The comparison behavior must meet the general definition of equivalence, i.e., it must be symmetric, reflexive, and transitive.

The primary motivations for providing user-defined (i.e., heterodox) operations on storage homes are the following:

- *to provide a means of exposing behaviors embedded in the store (e.g., stored procedures in a relational database)*
- *to allow federation of stores*

Polymorphism in storage home interfaces may be particularly useful for federating stores, or more precisely, storage homes on different stores.

6.8.8 StorageHomeBase

The **StorageHomeBase** interface is defined by the following IDL specification:

```
module Components {
module Persistence {

    typedef sequence<octet> PersistentID;
    typedef string PSSTypeId;

    exception DoesNotExist {};

    local StorageHomeBase {

        TypeId managed_storage_type_id();

        IncarnationBase incarnate(in PersistentID pid)
        raises (DoesNotExist);

        void remove(in IncarnationBase inc) raises (DoesNotExist);
        void remove_by_pid(in PersistentID pid) raises (DoesNotExist);

        void flush() raises (PersistentStoreError);
        void refresh() raises (PersistentStoreError);

    };

    local KeylessStorageHomeBase : StorageHomeBase {
        PersistentID create_pid();
    };
};
};
```

incarnate

This operation produces an incarnation for the storage object identified by the specified PID value. If the PID value does not denote an existing storage object within the persistent store, the operation raises a **DoesNotExist** exception.

remove

This operation removes the storage object associated with the specified incarnation. Subsequent attempts to incarnate the object shall raise a **DoesNotExist** exception. Subsequent attempts to invoke operations on the incarnation shall raise an **OBJECT_NOT_EXIST** system exception. If the specified incarnation is not associated with an existing storage object in the persistent store, the operation shall raise a **DoesNotExist** exception.

remove_by_pid

This operation removes the storage object associated with the specified PID value. Subsequent attempts to incarnate the object shall raise a **DoesNotExist** exception. If the specified PID is not associated with an existing storage object in the persistent store, the operation shall raise a **DoesNotExist** exception.

flush

If the current transaction policy of the persistent store to which the target storage home belongs is **NonTransactionalAccess**, the **flush** operation makes durable all of the modifications to active incarnations managed by the target storage home, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store to which the target storage home belongs is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, **flush** behaves as follows:

- If the invoking thread is associated with a transaction context, flush makes durable all state modifications made in the current transactional scope for incarnations managed by the target storage home, flushing them to the underlying store.
- If the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to reconcile the changes and make them durable, then the **PersistentStoreError** exception is thrown.

refresh

If the current transaction policy of the persistent store to which the target storage home belongs is **NonTransactionalAccess**, the **refresh** operation refreshes the state of all active incarnations managed by the target storage home with the current durable state from the underlying store, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store to which the target storage home belongs is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, **refresh** behaves as follows:

- If the invoking thread is associated with a transaction context, **refresh** guarantees that, subsequent to the **refresh** invocation, the first time a thread in the same transactional context as the caller of **refresh** accesses or modifies an incarnation managed by the target storage home, the state of that incarnation will be refreshed before the access is allowed.
- If the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to refresh the appropriate incarnations, the **PersistentStoreError** exception is thrown.

6.8.9 *KeylessStorageHomeBase*

The **KeylessStorageHomeBase** interface is defined by the following IDL specification:

```
module Components {
module Persistence {

    typedef sequence<octet> PersistentID;

    local KeylessStorageHomeBase : StorageHomeBase {
        PersistentID create_pid();
    };
};
};
```

create_pid

This operation creates a new storage object and allocates a PID value for the it. The PID value is returned. An incarnation of the new storage object may be obtained by invoking **StorageHomeBase::incarnate** or the type-specific **incarnate_by_pid** operation on the storage home.

6.9 Persistent store

A persistent store binds a specified set of storage homes to an underlying storage mechanism. A persistent store mediates between the behavior and form of the underlying storage mechanism and the abstract semantics offered by storage homes and incarnations, encapsulating and managing a set of sessions that connected the persistent store with the underlying storage mechanism.

A persistent store defines a scope of identity for storage homes and storage objects, in that PIDs are unique within a persistent store. A persistent store also defines a scope of reference for storage objects, in that a storage object may only hold a references for storage objects in the same persistent store.

6.9.1 Syntax

The syntax for declaring a persistent store is as follows:

<persistent_store> ::= <persistent_store_header> <persistent_store_body>

<persistent_store_header> ::= “persistentStore” <identifier>

<persistent_store_body> ::= “{” <persistent_store_member>+ “}”

<persistent_store_member> ::= <storage_home_dcl> “;”
| <local_operation> “;”

<storage_home_dcl> ::= “provides” <storage_home_type_spec>
<simple_declarator>

<storage_home_type_spec> ::= <scoped_name>

A persistent store definition consists of the following elements:

- the keyword **persistentStore**
- an <*identifier*> that names the persistent store type in the enclosing module scope
- a body containing one or more members, where each member is either:
  - a <*storage_home_dcl*> that declares a storage home offered by the persistent store, or
  - a local operation declaration

6.9.2 Equivalent local interfaces

The language mappings for persistent store definitions are described in terms of equivalent local interfaces. Given a persistent store definition of the following form:

persistentStore *<store_name>* {*<contents>*};

The equivalent local interface would have the form:

```
local <store_name>
: Components::Persistence::PersistentStoreBase {
    <equivalent_contents>
};
```

The base class **Components::Persistence::PersistentStoreBase** is described in Section 6.9.5, “PersistentStoreBase interface.

6.9.3 Obtaining storage homes from a persistent store

Given a <*storage_home_dcl*> of the following form in a persistent store definition:

provides *<storage_home_type>* *<name>*;

The equivalent operation on the store’s equivalent local interface would have the following form:

***<storage_home_type>* provide_*<name>*();**

This operation will return a local reference to a storage home of <*storage_home_type*>, the type specified in the <*storage_home_dcl*>. All storage objects managed by that storage home are associated with the target persistent store.

Storage homes are identified relative to their owning persistent store by the name that constitutes the <simple_declarator> in the **provides** declaration in the persistent store definition. Storage homes can also be obtained with the **PersistentStoreBase::provide_storage_home** operation, specifying the name as the operation parameter.

6.9.4 Local operations on persistent stores

Local operations in persistent store definitions have the same precise form in the persistent store's equivalent local interface. Local operations must be implemented by the user, either by programmatically extending a skeleton generated by the CIF provider from the CIDL specification of the store, or with unspecified tools provided by the CIF provider.

> *It is expected that CIF providers will expose proprietary, storage-specific APIs on their implementations of PersistentStoreBase, and that user-provided implementations of local operations on the persistent store will employ these proprietary interfaces.*

Language mappings must specify skeletons for type-specific persistent store definitions that allows users to implement any local operations defined on the store.

6.9.5 PersistentStoreBase interface

The **PersistentStoreBase** interface exposes the basic behavioral capabilities shared by all type-specific persistent stores. A persistent store performs the following functions:

- provide storage homes of the types specified in the persistent store's definition
- mediate operations between the CIF interfaces supplied to an application (e.g., incarnations, storage homes, etc.) and an underlying storage engine in a way that provides the semantics required by this specification, including:
  - maintaining correct, durable logical state of storage objects
  - exposing correct views of storage object logical stage of through incarnations
  - managing storage object life cycles
  - maintaining assocations between storage objects, PIDs, and primary key values
  - guaranteeing appropriate ACID properties of transactions

These behaviors are provided by the CIF vendor, in the form of an implementation of **PersistentStoreBase**. Specific persistent store types defined in CIDL extend this basic mechanism by inheritance, adding accessors for declared storage homes and implementing user-defined operations on the store.

The **PersistentStoreBase** interface is defined by the following IDL specification:

```
module Components {
module Persistence {

    struct Property {
        string name;
        any    value;
    };

    typedef sequence<Property> PropertyList;

    typedef sequence<octet> PersistentId;
    typedef sequence<PersistentId> PersistentIdList;


    local PersistentStoreBase {

        void open(in string name, in PropertylList params)
        raises(NoPermission);

        void close();

        StorageHomeBase provide_storage_home(in string home_name)
        raises(NotFound);

        void flush() raises(PersistentStoreError);

        void flush_by_pids(in PersistentStoreIdList pids)
        raises(PersistentStoreError);

        void refresh() raises(PersistentStoreError);

        void refresh_by_pids(in PersistentStoreIdList pids)
        raises(PersistentStoreError);


    };
};
};
```

open

This operation opens the target persistent store, associating it with a particular underlying store. The **name** parameter denotes the store (e.g., a particular data base) to which the persistent store will mediate access and updates. The form and meaning of the value of **name** are product-specific. The **params** parameter contains a list of name-value pair properties. The use of this parameter is product-specific. It is intended to allow the caller to define the required behaviors and characteristics of the underlying storgage enginer, or to provide additional information required by the product to open the underlying store. The **Components::Persistence::NoPermission** exception is raised if the user does not

have permissions to open the specifed storage. A persistent store must be opened before it can be used. If any operation other than open is invoked before the persistent store is opened, the **BAD_INV_ORDER** system exception is raised.

provide_storage_home

The operation **provide_storage_home** returns the storage home denoted by the home_name parameter. The value of this parameter must be the name of a storage home provided by the persistent store, as declared in the persistent store's SDL definition. The **NotFound** exception is raised if the name parameter does not denote a storage home provided by the persistent store.

close

This operation disassociates the persistent store from the underlying store. Any uncommitted transactions at the time of closure are marked for rollback. Cached state may be flushed, if the implementation of the persistent store deems it logically appropriate. The invocation of any operation other than open on the persistent store after it is closed will raise the **BAD_INV_ORDER** system exception. Any invocation on a storage home or an incarnation provided by the persistent store after the store is closed will raise an **INV_OBJREF** system exception.

flush

If the current transaction policy of the persistent store is **NonTransactionalAccess**, the **flush** operation makes durable all of the modifications to active incarnations managed by persistent store, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store to is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, **flush** behaves as follows:

- If the invoking thread is associated with a transaction context, flush makes durable all state modifications made in the current transactional scope for incarnations managed by persistent store, flushing them to the underlying store.
- If the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to reconcile the changes and make them durable, then the **PersistentStoreError** exception is thrown.

flush_by_pids

If the current transaction policy of the persistent store is **NonTransactionalAccess**, the **flush** operation makes durable all of the modifications to active incarnations whose PIDs are contained in the **pids** parameter, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store to is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, **flush_by_pids** behaves as follows:

- If the invoking thread is associated with a transaction context, flush makes durable all state modifications made in the current transactional scope for incarnations whose PIDs are contained in the **pids** parameter, flushing them to the underlying store.
- If the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to reconcile the changes and make them durable, then the **PersistentStoreError** exception is thrown.

refresh

If the current transaction policy of the persistent store is **NonTransactionalAccess**, the **refresh** operation refreshes the state of all active incarnations managed by the persistent store with the current durable state from the underlying store, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, and the invoking thread is associated with a transactional context, denoted by T_i, **refresh_by_pids** causes the following behavior:

- the first time any thread in the transactional context T_i touches (i.e., invokes an accessor or mutator) an incarnation managed by the persistent store prior to teh **refresh** invocation, the state of that incarnation will have been refreshed prior to the **refresh** invocation before the access is allowed. This is true for all incarnations touched by threads in transactional context T_i.

If the current transaction policy of the persistent store is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, and the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to refresh the appropriate incarnations, the **PersistentStoreError** exception is thrown.

refresh_by_pids

If the current transaction policy of the persistent store is **NonTransactionalAccess**, the **refresh_by_pids** operation refreshes the state of all active incarnations whose PIDs appear in the **pids** parameter with the current durable state from the underlying store, regardless of the transactional context of the calling thread.

If the current transaction policy of the persistent store is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, and the invoking thread is associated with a transactional context, denoted by T_i, **refresh_by_pids** causes the following behavior:

- the first time any thread in the transactional context T_i touches (i.e., invokes an accessor or mutator) an incarnation whose PID appears in the **pids** parameter prior to the **refresh** invocation, the state of that incarnation will have been refreshed prior to the **refresh** invocation before the access is allowed. This is true for all incarnations touched by threads in transactional context T_i.

If the current transaction policy of the persistent store is **LocalTransactionalAccess** or **DistributedTransactionalAccess**, and the invoking thread is not associated with a transactional context, the **NoTransaction** exception is raised.

If the persistent store implementation is unable to refresh the appropriate incarnations, the **PersistentStoreError** exception is thrown.

6.9.6 *GenericPersistentStore*

The **GenericPersistentStore** interface allows storage homes to be provided to an application without the need for an explicitly-defined persistent store type. PSS providers shall supply an implementation of **GenericPersistentStore** and make it available through the persistent store factory supplied with their product. The means by which specified storage home types are exposed through the **GenericPersistentStore** interface is unspecified.

The **GenericPersistentStore** interface is defined by the following IDL specification:

```
module Components {
module Persistence {

    local GenericPersistentStore
    : PersistentStoreBase {

        StorageHomeBase provide_storage_home_by_type(
            in CORBA::RepositoryId type_id
        ) raises (NotFound);

    };
};
```

The **provide_storage_home_by_type** operation returns a storage home of the type specified by the type_id parameter. If no storage home of the requested type is available, the **NotFound** execption is raised.

The Container Programming Model 7

This chapter describes the **container programming model** offered for CORBA components.The container is the server's runtime environment for a CORBA component implementation. This environment is implemented by a deployment platform such as an application server or a development platform like an IDE. A deployment platform typically provides a robust execution environment designed to support very large numbers of simultaneous users. A development platform would provide enough of a runtime to permit customization of CORBA components prior to deployment but perhaps support a limited number of concurrent users. From the point of view of the CORBA component implementation, such differences are "qualities of service" characteristics and have no effect on the set of interfaces the component implementor can rely on. This chapter is organized as follows:

- Section 7.2 on page 137 introduces the programming model and defines the elements that comprise it.

  The container programming model is an API framework designed to simplify the task of building a CORBA application. Although the framework does not exclude the component developer from using any function currently defined in CORBA, it is intended to be complete enough in itself to support a broad spectrum of applications.

- Section 7.3 on page 140 describes the programming model the component implementor is to follow.

  The programming model identifies the architectural choices which must be made to develop a CORBA component which can be deployed in a container.

- Section 7.4 on page 153 describes the interfaces seen by the component developer.

  These interfaces constitute the contract between the container provider and the component implementor. Together with the client programming interfaces defined in Chapter 5 which can be used by servers as well as clients, they define the server programmer's API.

- Section 7.5 on page 174 describes the client view of a CORBA component.

The **client programming model** as defined by the IDL extensions has been described previously (Chapter 5). This section describes the specific use of CORBA required by a client, which is **NOT** itself a CORBA component, of a CORBA component written to the server programming model described in Section 7.4 on page 153.

7.1 Change History

The following changes were made in the orbos/99-02-01 version of the document:

1. Container APIs to support persistence have been added.
2. Some interfaces and operations have been adjusted to more closely align with EJB. In addition rationale text has been added to contrast the component APIs with the EJB APIs and explain the differences where they exist.
3. Container-managed persistence has been added for **process** components. It previously was supported only for **entity** components.
4. Multiple servant lifetime policies are now supported for **entity** components. **Entity** components were previously restricted to only the **transaction** servant lifetime policy.
5. Interface names have been synchronized with those defined in the abstract model (Chapter 5) and **Cookies** have replaced **Tokens** for consistency.
6. Client programming examples have been changed from pseudo-code to Java.
7. Locality-constrained interfaces previously defined as **valuetype** have been changed to use the new **local** keyword.
8. The **set_timeout** operation has been added to **Transaction**. It was inadvertently omitted in previous versions.
9. IDL identifier names were changed to conform to the OMG IDL Style Guide (ab/98-06-03).
10. Miscellaneous clarifications have been made to the text.

The following changes have been made since orbos/99-02-01:

1. An introduction section was added to summarize the key elements of the programming model as seen by the component developer, the component client, and the container provider.
2. The client programming model section was moved to the end of the chapter since it depends on the server programming model and is not a normative description of what the client ORB must do.
3. The server programming model was expanded to more clearly identify which data comes from which source and what effect it has on the container's behavior at runtime.

4. The event model was updated to include both shared notification channels and dedicated notification channels
5. The persistence section was elaborated to more clearly describe the various persistence choices supported for CORBA components.
6. Added a new component category called **session**, which supports transient state with multiple servant lifetime policies similar to an MTS component, has been added. The previous **session** component has been renamed to **service** which more accurately describes its properties.
7. We've made a lot more rationale text into the rationale font.
8. Miscellaneous clarifications have been made to the text.

All changes are clearly marked with change bars. In general existing text which was moved will not have change bars.

7.2 Introduction

The container programming model is made up of several elements:

- The **external types** which define the interfaces available to a component client
- The **container type** which defines the API framework used by the component developer
- The **container implementation type** which defines the interactions between the container and the rest of CORBA (including the POA, the ORB and the CORBA services)
- The **component category** which is the combination of the **container type** (i.e. the server view) and the **external types** (i.e. the client view)

The overall architecture is depicted in Figure 7-1 below::

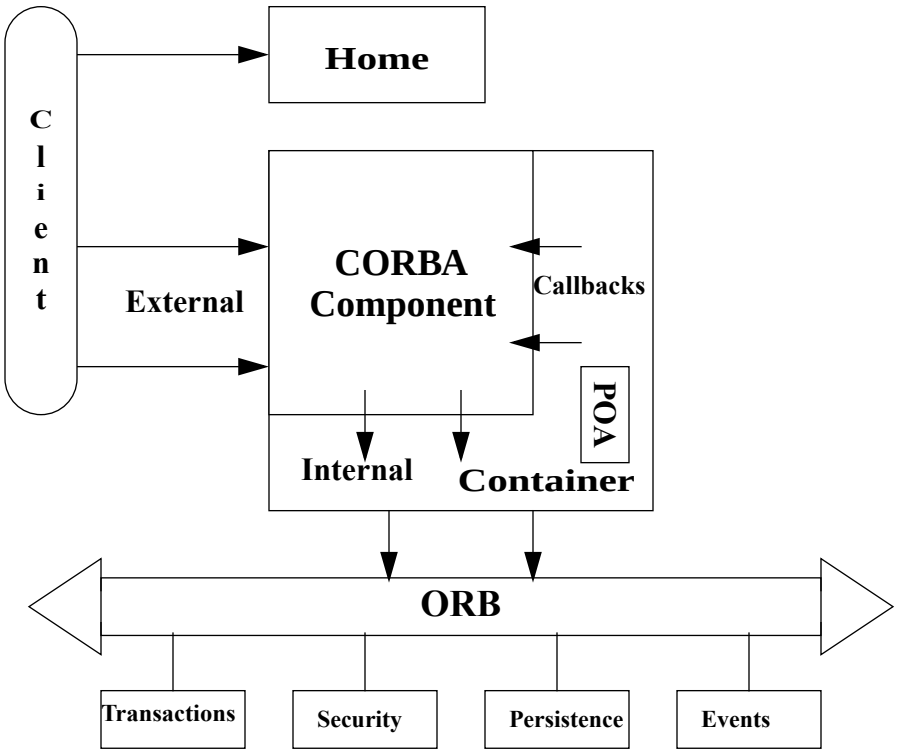

Figure 7-1 The Architecture of the Container Programming Model

The **external types** are defined by the component IDL including the home specification. These interfaces are righteous CORBA objects and are stored in the Interface Repository for client use.

The **container type** is a framework made up of internal interfaces and callback interfaces used by the component developer. These are defined using the new **local** keyword in IDL for specifying locality-constrained interfaces. The **container type** is selected using CIDL which describes component implementations.

The **container implementation type** is controlled by policies which specify distinct interaction patterns with the POA and a set of CORBA services. These are defined using XML in the component descriptor and used by the container factory to create a POA when the container is created.

The **component category** is a specific combination of **external types** and **container type** used to implement an application with the CORBA component technology.

7.2.1 External Types

The **external types** of a component are the contract between the component developer and the component client. We distinguish between two forms of **external types**: the **home** interface and the **application** interfaces. Home interfaces support operations which allow the client to obtain references to one of the application interfaces the component implements. From the client's perspective, two design patterns are supported - factories for creating new objects and finders for existing objects. These patterns are distinguish by the presence of a **primaryKey** parameter in the home IDL declaration.

- A home interface with a **primaryKey** declaration supports finders and is a **PrimaryKeyVisibility** client.
- A home interface without a **primaryKey** declaration does not support finders and is a **NoKeyVisibility** client. All home types support factory operations.

These **external types** are righteous CORBA objects which are seen as remote interfaces by a client. They are defined in IDL and represented in the Interface Repository.

> *These are analogous to the* **EJBHome** *and* **EJBObject** *interfaces of Enterprise Java Beans.*

7.2.2 Container Type

A **container type** defines the API framework between the component and its container. This specification defines two distinct base types which define the common APIs and a set of derived APIs which provide additional function. The **transient container type** defines a framework for components using transient object references. The **persistent container type** defines a framework for components using persistent object references.

7.2.3 Container Implementation Type

A **container implementation type** specifies the required interaction pattern between the container, the POA and the CORBA services. We define three interaction patterns as part of this specification:

- **stateless** - which uses transient object references in conjunction with a POA servant which can support any **ObjectId**
- **conversational** - which uses transient references in conjunction with a POA servant that is dedicated to a specific **ObjectId**
- **durable** - which uses persistent references in conjunction with a POA servant that is dedicated to a specific **ObjectId**

> *It should be obvious that the fourth possibility (persistent references with a POA servant that can support any ObjectId) makes no sense and is therefore not included.*

7.2.4 Component Categories

The **component categories** are defined as the valid combinations of **external types**, **container type**, and **container implementation type**. The following table should make this clear:

Table 7-1 Definition of the Component Categories

Container Implementation Type	Container Type	Primary Key	Component Categories	EJB Bean Type
stateless	transient	No	Service	-
conversational	transient	No	Session	Session
durable	persistent	No	Process	-
durable	persistent	Yes	Entity	Entity

7.3 The Server Programming Environment

The component container provides interfaces to the component. These interfaces support access to CORBA services (transactions, security, notification, and persistence) and to other elements of the component model. This section describes the features of the container which are selected by the deployment descriptor packaged with the component implementation. These features comprise the design decisions to be made in developing a CORBA component. Details of the interfaces provided by the container are provided in Section 7.4 on page 153.

7.3.1 Component Containers

Containers provide the run-time execution environment for CORBA components. A container is a **framework** for integrating transactions, security, events, and persistence into a component's behavior at runtime. A container provides the following functions for its component:

- all component instances are created and managed at runtime by its container
- containers provide a standard set of services to a component, enabling the same component to be hosted by different container implementations

Components and homes are deployed into containers with the aid of container specific tools. These tools generate additional programming language and metadata artifacts needed by the container. The tools provide the following services:

- editing the configuration metadata
- editing the deployment metadata
- generating the implementations needed by the containers to support the component

The container framework defines two forms of interfaces:

- **Internal interfaces** - These are locality-constrained interfaces defined as **local** types which provide container functions to the CORBA component.

  These are similar to the **EJBContext** *interface in Enterprise Java Beans.*

- **Callback interfaces** - These are also **local** types invoked by the container and implemented by a CORBA component.

  These interfaces provide functions analogous to the **SessionBean** *and* **EntityBean** *interfaces defined by Enterprise Java Beans.*

This architecture is depicted in Figure 7-1 on page 138.

We define a small set of **container types** to support a broad spectrum of component behavior with their associated **internal** and **callback** interfaces as part of this specification. These **container types** are defined using the new **local** keyword in IDL introduced in Section 4.1 on page 19 for specifying locality-constrained interfaces.

Additional component behavior is controlled by policies specified in the deployment descriptor.This specification defines policies which support POA interactions (**container implementation type**), servant lifetime management, transactions, security, events, and persistence. See the deployment chapter (Chapter 9), specifically Section 9.4 on page 243, for details of how container policies are specified.

CORBA containers are designed to permit their use as Enterprise Java Beans containers. This allows a CORBA infrastructure to be the foundation of EJB, enabling a more robust implementations of the EJB specification. To support EJBs natively within a CORBA container, the container must support both sets of APIs. This architecture is defined in Chapter 11 of this specification.

7.3.2 Container Implementation Type

The CORBA Component Specification defines a set of **container implementation types** which create either **TRANSIENT** or **PERSISTENT** object references and use either a 1:1 or 1:N mapping of **Servant** to **ObjectId**. These **container implementation types** are summarized in Table 7-2 below. A given component implementation supports one and only one **container implementation type**.

Table 7-2 Container Implementation Type Definitions

Container Implementation Type	Object Reference	Servant:OID Mapping
stateless	**TRANSIENT**	1:N
conversational	**TRANSIENT**	1:1
durable	**PERSISTENT**	1:1
(Invalid)	**PERSISTENT**	1:N

A **container implementation type** is specified using CIDL and is used to either create or select a component container at deployment time.

7.3.2.1 Component References

TRANSIENT objects support only the factory design pattern. They are created by operations on the home interface defined in the **component** declaration.

PERSISTENT objects support either the factory design pattern or the finder design pattern, depending on the **component category**. **PERSISTENT** objects support **component-managed** or **container-managed** persistence. **PERSISTENT** objects can be used with CORBA persistence or any user-defined persistence mechanism. When CORBA persistence is used, servant management is aligned with a **PersistentId** as defined in Chapter 6 and the container supports the transformation of an **ObjectId** to and from a **PersistentId**. A **PersistentId** provides a persistent handle for a class of objects whose permanent state resides in a persistent store (e.g. a database).

References are exported for client use by registering them with a **HomeFinder** which the client subsequently interrogates. The finder design pattern may also export references by binding them to the CORBA naming service in the form of externally visible names.

7.3.2.2 Servant to ObjectId Mapping

Component implementations may use either the 1:1 or 1:N mapping of **Servant** to **ObjectId** with **TRANSIENT** references (**stateless** and **conversational container implementation type**, respectively) but may use only the 1:1 mapping with **PERSISTENT** references.

- A 1:N mapping allows a **Servant** to be shared among all requests for the same interface and therefore requires the object to be stateless (i.e. it has no identity).
- A 1:1 mapping binds a **Servant** to a specific **ObjectId** for an explicit servant lifetime policy (see Section 7.3.5 on page 143) and therefore is stateful.

7.3.2.3 Threading Considerations

CORBA components support two threading models: **serialize** and **multithread**. A threading policy of **serialize** means that the component implementation is not thread safe and the container will prevent multiple threads from entering the component simultaneously. A threading policy of **multithread** means that the component is capable of mediating access to its state without container assistance and multiple threads will be allowed to enter the component simultaneously. Threading policy is specified in the component's deployment descriptor.

> *A threading policy of* **serialize** *is required to support an EJB since EJB's are defined to be single-threaded.*

7.3.3 Component Factories

A home is a component factory, responsible for creating instances of all interfaces exported by a component. Factory operations are defined on the home interface using the **factory** declaration. A default factory is automatically defined whose

implementation may be generated by tools using the information provided in the **component** IDL. Specialized factories (e.g. factories that accept user-defined input arguments) must be implemented by the component developer. Factory operations are typically invoked by clients but may also be invoked as part of the implementation of the component. A CORBA component implementation can locate its home interface using an interface provided by the container.

7.3.4 Component Activation

CORBA components rely on the automatic activation features of the POA to tailor the behavior of the components using information present in the component's deployment descriptor. Once references have been exported, clients make operation requests on the exported references. These requests are then routed by the ORB to the POA that created the reference and then the component container. This enables the container to control activation and passivation for components, apply policies defined in the component's descriptor, and invoke callback interfaces on the component as necessary.

7.3.5 Servant Lifetime Management

Servants are programming language objects which the POA uses to dispatch operation requests based on the **ObjectId** contained in the object key. The server programming model for CORBA components includes facilities to efficiently manage the memory associated with these programming objects. To implement this sophisticated memory management scheme requires the server programmer to make several design choices:

- The **container type** must be chosen.
- The **container implementation type** must be chosen.
- A servant lifetime policy is selected. CORBA components support four servant lifetime policies (**method**, **transaction**, **component**, and **container**).
- The designer is required to implement the callback interface associated with his choice.

The servant lifetime policies are defined as follows:

method

The **method** servant lifetime policy causes the container to activate the component on every operation request and to passivate the component when that operation has completed. This limits memory consumption to the duration of an operation request but incurs the cost of activation and passivation most frequently.

transaction

The **transaction** servant lifetime policy causes the container to activate the component on the first operation request within a transaction and leave it active until the transaction completes and which point the component will be passivated. Memory remains allocated for the duration of the transaction.

component

The **component** servant lifetime policy causes the container to activate the component on the first operation request and leave it active until the component implementation requests it to be passivated. After the operation which requests the passivation completes, the component will be passivated by the container. Memory remains allocated until explicit application request.

container

The **container** servant lifetime policy causes the container to activate the component on the first operation request and leave it active until the container determines it needs to be passivated. After the current operation completes, the component will be passivated by the container. Memory remains allocated until the container decides to reclaim it.

The following table shows the relationship between the **container implementation type**, the **container type**, and the servant lifetime policies:

Table 7-3 Servant Lifetime Policies by Container Type

Container Implementation Type	Container Type	Valid Servant Lifetime Policies
stateless	transient	method
conversational	transient	method, transaction, component, container
durable	persistent	method, transaction, component, container

7.3.6 Transactions

Transaction policies are defined in the component's deployment descriptor. The container uses these descriptions to make the proper calls to the CORBA transaction service. The transaction policy defined in the component's deployment descriptor is applied by the container prior to invoking the operation.

The following table summarizes the effect of the various transaction policy attributes and the presence or absence of a client transaction on the transaction which is used to invoke the requested operation on the component:

Table 7-4 Effects of Transaction Policy Attribute

Transaction Attribute	Client Transaction	Component's Transaction
NOT_SUPPORTED	-	-
	T1	-
REQUIRED	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1

Table 7-4 Effects of Transaction Policy Attribute

Transaction Attribute	Client Transaction	Component's Transaction
REQUIRES_NEW	-	T2
	T1	T2
MANDATORY	-	EXC (**TRANSACTION_REQUIRED**)
	T1	T1
NEVER	-	-
	T1	EXC (**INVALID_TRANSACTION**)

not_supported

This component does not support transactions. If the client does not provide a current transaction the operation is invoked immediately. If the client provides a current transaction, it is suspended (**CosTransactions::Current::suspend**) before the operation is invoked and resumed (**CosTransactions::Current::resume**) when the operation completes.

required

This component requires a current transaction to execute successfully. If one is supplied by the client, it is used to invoke the operation. If one is not provided by the client, the container starts a transaction (**CosTransactions::Current::begin**) before invoking the operation and attempts to commit the transaction (**CosTransactions::Current::commit**) when the operation completes.

supports

This component will support transactions if one is available. If one is provided by the client, it is used to invoke the operation. If one is not provided by the client. the operation is invoked immediately.

requires_new

This component requires its own transaction to execute successfully. If no transaction is provided by the client, the container starts one (**CosTransactions::Current::begin**) before invoking the operation and tries to commit it (**CosTransactions::Current::commit**) when the operation completes. If a transaction is provided by the client, it is first suspended (**CosTransactions::Current::suspend**), a new transaction is started (**CosTransactions::Current::begin**), the operation invoked, the component's transaction attempts to commit (**CosTransactions::Current::commit**), and the client's transaction is resumed (**CosTransactions::Current::resume**).

mandatory

The component requires that the client be in a current transaction before this operation is invoked. If the client is in a current transaction, it is used to invoke the operation. If not, the **TRANSACTION_REQUIRED** exception is raised.

never

This component requires that the client not be in a current transaction to execute successfully. If no current transaction exist, the operation is invoked. If a current transaction exists, the **INVALID_TRANSACTION** exception is raised.

EJB has all of the above transaction policies except for "never." Their definition in EJB is identical to their definition in CORBA Components. The "never" policy is added for completeness. EJB also has a bean-managed transaction policy which requires policing the transaction APIs in the container. Rather than add that directly to the CORBA component model, we inject the enforcement code into the glue required to make an EJB run in a CORBA container.

7.3.7 Security

Security policy is applied consistently to all categories of components. The container extracts the requested security policy from the deployment descriptor, checks the active credentials for invoking operations, and, if necessary adjusts the credentials to accommodate the requested policy. The security policy remains in effect until changed by a subsequent invocation on a different component having a different policy.

7.3.8 Events

CORBA components use a simple subset of the CORBA notification service to emit and consume events. The subset can be characterized by the following attributes:

- Events are represented as **valuetypes** to the component implementor and the component client
- The event data structure is mapped to an **any** in the body of a structured event presented to and received from CORBA notification.
- The fixed portion of the structured event is added to the event data structure by the container on emitting and removed from the event data structure when consuming
- Components support two forms of event generation and consumption
  - a component may be an exclusive supplier or an exclusive consumer of a given type of event.
  - a component may supply or consume events from a channel that other CORBA notification users are also utilizing
- A CORBA component emits events using the push model.
- A CORBA component consumes events using the push model.
- Events may have transactional behavior depending on both **container implementation type** and policy data in the deployment descriptor.
- All channel management is implemented by the container, not the component.
- Filters are set administratively by the container, not the component
- Quality of service characteristics are specified in the deployment descriptor.

Because events can be emitted and consumed by clients as well as component implementations, the operations for emitting and consuming events are generated from the specifications in component IDL The container is responsible for mapping these operations to the CORBA notification service to provide a robust event distribution network.

Quality of service policies are defined for both events to be emitted and for events to be consumed. The policies must be valid for the **container implementation type** the component is to be deployed in. The possible values are as follows:

normal

A **normal** event policy indicates the event should be emitted or consumed outside the scope of a transaction. If a current transaction is active, it is suspended before sending the event or invoking the operation on the proxy object provided by the component.

default

A **default** event policy indicates the event should be emitted or consumed regardless of whether a current transaction exists. If a current transaction is active, the operation is transactional. If not, it is non-transactional

transaction

A **transaction** policy indicates the event should be emitted or consumed within the scope of a transaction. If a current transaction is not active, a new one is initiated before sending the event or invoking the operation on the proxy object provided by the component. The new transaction is committed as soon as the operation is complete.

7.3.9 Persistence

A **persistent container type** supports the use of a persistence mechanism for making component state durable, e.g. storing it in a persistent store like a database. A **persistent container type** defines two forms of persistence support:

- **container-managed persistence** where the component developer simply defines the state which is to be made persistence and the container (in conjunction with generated code) automatically saves and restores state as required.

  Container-managed persistence is selected by defining the abstract state associated with a component segment using the **storage** declaration defined in Chapter 6 and connecting that **storage** to a component segment using CIDL.

- **component-managed persistence** where the component developer assumes the responsibility of saving and restoring state when requested to do so by the container.

  Component-managed persistence is selected via CIDL declaration and triggered by the container invoking the callback interfaces defined later in this chapter (Section 7.4 on page 153) which the component must implement.

Container-managed persistence may be accomplished using the CORBA persistence mechanism or any user-defined persistence mechanism. When the CORBA persistence mechanism is used, it is possible to provide automatic code generation for the storage factories, finders, and some callback operations.

If container-managed persistence is to be accomplished with a user-defined persistence mechanism, the component developer must implement the various persistence classes defined in the persistence framework (see Chapter 6 for more details).

Component-managed persistence is also supported by the **persistent container type**. Like container-managed persistence, the component developer has two choices: to use the CORBA persistence mechanism or some user-defined persistence mechanism. But since no declarations are available to support code generation, the component developer is responsible for implementing both the callback interfaces and the persistence classes. The container may, but is not obliged to, provide the component with a connection to a persistent store, but no standard way to do this is defined in this specification.

Table 7-5 below summarizes the choices and their required responsibilities:

Table 7-5 Persistence Support for persistent **container type**

Persistence Support	Persistence Mechanism	Responsibility	Persistence Classes	Callback Interfaces
Container Managed	CORBA	Container	Generated Code	Generated Code
Container Managed	User	Container	Component implements	Generated Code
Component Managed	CORBA	Component	Generated Code	Component implements
Component Managed	User	Component	Component implements	Component implements

7.3.10 Application Operation Invocation

The application operations of a component are specified on either the component's supported interface or one of the provided interfaces. Since CORBA components support multiple interfaces, these operations are normal CORBA object invocations.

> *In Enterprise Java Beans, all remote invocations are made on the* **EJBObject** *interface which intercepts the object-dispatch and delegates application operation invocations to a particular bean instance. CORBA components support multiple interfaces eliminating the need for delegation, and use the facilities of the POA to intercept object dispatch. This eliminates the need for an equivalent concept in CORBA components, reducing the number of artifacts which need to be generated, installed, and activated/passivated.*

Application operations may raise exceptions, both application exceptions (i.e. those defined as part of the IDL interface definition) and system exceptions (those that are not). Exceptions defined as part of the IDL interfaces defined for a component (that includes both provided interfaces and supported interfaces) are raised back to the client directly and do not affect the current transaction. All other exceptions raised by the application are intercepted by the container which then raises the **TRANSACTION_ROLLEDBACK** exception to the client, if a transaction is active. Otherwise they are reported back to the client directly.

7.3.11 Component Implementations

A component implementation consists of one or more **executors** as described in Chapter 6. Each **executor** describes the implementation characteristics of a particular component segment. The transient **container type** consists of a single **executor** with a single segment which is activated in response to an operation request on any component interface. The persistent **container type** can be made up of multiple segments, each of which is associated with a different **storage**. Each segment is independently activated when an operation request on an interface associated with that segment is received.

7.3.12 Component Categories

As indicated in Section 7.2.4 on page 140, this specification defines four **component categories** whose behavior is specified by the two **container types**. Additionally we reserve a **component category** to describe the empty container (i.e. a **container type** which does not use one of the API frameworks defined in this specification). The four **component categories** are described briefly in the following sections:

7.3.12.1 The Service Component

The **service** component is a CORBA component with the following properties:

- no state
- no identity
- behavior

The lifespan of a **service** component is equivalent to the lifetime of a single operation request (i.e. **method**) so it is useful for functions such as command objects which have no persistence beyond the lifetime of a single client interaction with them. A **service** component can also be compared to a traditional TP monitor program like a Tuxedo service or a CICS transaction. A **service** component provide a simple way of wrapping existing procedural applications.

> *At first glance, a* **service** *component looks like a stateless EJB Session-Bean, however we do not model it that way so that we can optimize the runtime performance of* **service** *components by reducing the number of callback operations supported. Instead we model an EJB stateless Session-Bean as a* **session** *component with a servant lifetime policy of* **method**.

The following table (Table 7-6) summarizes the characteristics of **service** component as seen by the server programmer:

Table 7-6 A service component design characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus **TransientOrigin**
Callback Interfaces	**ServiceComponent**
Container Implementation Type	stateless
External Types	NoKeyVisibility
Client Design Pattern	Factory
Persistence	No
Servant Lifetime Policy	method
Transactions	May use, but not included in current transaction
Events	Transactional or Non-transactional
Executor	Single segment with a single servant and no storage

Because of its absence of state, any programming language servant can service any **ObjectId**, enabling such servants to be managed as a pool or dynamically created as required, depending on usage patterns. Because a **service** component has no identity, **ObjectIds** can be managed by the POA, not the component implementor, and the client sees only the factory design pattern.

7.3.12.2 The Session Component

The **session** component is a CORBA component with the following properties:

- transient state
- identity which is not persistent
- behavior

The lifespan of a **session** component is specified using the servant lifetime policies defined in Section 7.3.5 on page 143. A **session** component (with a **transaction** lifetime policy) is similar to an MTS component and is useful for modeling things like iterators, which require transient state for the lifetime of a client interaction but no persistent store.

> *The* **session** *component is used to implement the SessionBean of EJB. Stateless SessionBeans have a servant lifetime policy of* **method** *and stateful SessionBeans have a servant lifetime policy of* **transaction** *or* **container**, *depending on whether transactions are used or not.*

The following table (Table 7-7) summarizes the characteristics of **session** component as seen by the server programmer:

Table 7-7 A Session Component Design Characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus TransientOrigin
Callback Interfaces	**SessionComponent** plus (optionally) **Synchronization**
container implementation type	conversational
Client Design Pattern	Factory
External Types	NoKeyVisibility
Persistence	No
Servant Lifetime Policy	Any
Transactions	May use, but not included in current transaction
Events	Transactional or Non-transactional
Executor	Single segment with a single servant and no storage

A programming language servant is allocated to an **ObjectId** for the duration of the servant lifetime policy specified. At that point, the servant can be returned to a pool and re-used for a different **ObjectId**. Alternatively, servants may be dynamically created as required, depending on usage patterns. Because a **session** component has no persistent identity, **ObjectIds** can be managed by the container, not the component implementor, and the client sees only the factory design pattern.

7.3.12.3 The Process Component

The **process** component is a CORBA component with the following properties:

- persistent state which is not visible to the client and is managed by the **process** component implementation or the container
- persistent identity which is managed by the **process** component and is made visible to the client only through user-defined operations
- behavior which may be transactional.

The **process** component is intended to model objects that represent business processes (e.g. applying for a loan, creating an order, etc.) rather than entities (e.g. customers, accounts, etc.). The major difference between **process** components and **entity** components is that the **process** component does not expose its persistent identity to the client (except through user-defined operations).

The **process** *component could be used to implement the stateful* **Session-Bean** *defined by EJB (which does not have identity) when its behavior is non-transactional. However, we choose to use the* **session** *component instead. When a* **process** *component exhibits transactional behavior, it is more like the* **EntityBean**.

The following table (Table 7-8) summarizes the characteristics of **process** component as seen by the server programmer:

Table 7-8 The Process Component Characteristics

Design Characteristic	Property
External Interfaces	As defined in component IDL
Internal Interfaces	Base set plus **PersistentOrigin**
Callback Interfaces	**PersistentComponent**
container implementation type	durable
Client Design Pattern	Factory
External Types	NoKeyVisibility
Persistence	Component-managed with or without PSS or Container-managed with or without PSS
Servant Lifetime Policy	Any
Transactions	May use, and can be included in current transaction
Events	Non-transactional or transactional events
Executor	Multiple segments with associated storage

A **process** component may have transactional behavior. The **persistent** container will interact with the CORBA transaction service to participate in the commit process.

The **process** component can use **container-managed** or **component-managed** persistence using either CORBA persistence or a user-defined persistence mechanism. The implications of the various choices are described in Section 7.3.9 on page 147. The persistent container uses callback interfaces which enable the **process** component's implementation to retrieve and save state data at activation and passivation, respectively.

7.3.12.4 The Entity Component

The **entity** component is a CORBA component with the following properties:

- persistent state which is visible to the client and is managed by the **entity** component implementation or the container
- identity which is architecturally visible to its clients through a **primaryKey** declaration

- behavior which may be transactional.

As a fundamental part of the architecture, **entity** components expose their persistent state to the client as a result of declaring a **primaryKey** value on their home declaration.

> *The* **entity** *component is used to implement the* **EntityBean** *in the Enterprise Java Beans specification.*

The following table (Table 7-9) summarizes the characteristics of **entity** component as seen by the server programmer:

Table 7-9 The Entity Component Characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base set plus **PersistentOrigin**
Callback Interfaces	**PersistentComponent**
container implementation type	durable
Client Design Pattern	Factory or Finder
External Types	PrimaryKeyVisibility
Persistence	Component-managed with or without PSS or Container-managed with or without PSS
Servant Lifetime Policy	Any
Transactions	May use, and can be included in current transaction
Events	Non-transactional or transactional events
Executor	Multiple segments with associated storage

The **entity** component can use **container-managed** or **component-managed** persistence using either CORBA persistence or a user-defined persistence mechanism. The implications of the various choices are described in Section 7.3.9 on page 147. The persistent container uses callback interfaces which enable the **entity** component's implementation to retrieve and save state data at activation and passivation, respectively.

7.4 Server Programming Interfaces

This section defines the **local** interfaces used and provided by the component developer. These interfaces are then grouped as follows:

- interfaces common to both **container types**
- interfaces supported by the transient **container type** only

- interfaces supported by the persistent **container type** only

Unless otherwise indicated, all of these interfaces are defined within the **Server** module embedded within the **Components** module (See appendix A.1 on page 399 for the proposed naming structure for CORBA 3.0 suggested by this specification).

7.4.1 Component Interfaces

All components deal with three sets of interfaces:

- **internal** interfaces which are used by the component developer and provided by the container to assist in the implementation of the component's behavior,
- **external** interfaces which are used by the client and implemented by the component developer, and
- **callback** interfaces which are used by the container and implemented by the component, either in generated code or directly, in order for the component to be deployed in the container.

A **container type** defines a base set of internal interfaces which the component developers use in their implementation. These interfaces are then augmented by others that are unique to the **component category** being developed.

- **ComponentContext** - which serves as a bootstrap and provides accessors to the other internal interfaces.

  Each **container type** has it's own specialization of **ComponentContext** which we refer to as a context.

- **ComponentId** - which masks the difference between references created by CORBA persistence and references created by other persistence mechanisms.

  Only the **persistent container type** supports the **ComponentId** interface.

- **BaseOrigin** - which supports operations for managing servant lifetime policy as well as for creating and managing object references in conjunction with the POA.

  Each **container type** has it's own specialization of **BaseOrigin** which we refer to as a context.

- **Transaction** - which wraps the demarcation subset of the CORBA transaction service required by the application developer.
- **Storage** - which wraps the persistence mechanism for both CORBA persistence and user-defined persistence mechanisms .
- **Security** - which wraps a subset of CORBA security to enable the application to validate requests against the credentials in effect for CORBA security.

All components implement a callback interface which is determined by the **component category**.

When a component instance is instantiated in a container, it is passed a reference to its context, a **local** interface used to invoke services.

The context interfaces serves the same role in CORBA components that the

> **EJBContext** *interface does in Enterprise Java Beans, viz. it provides the component implementation with access to the runtime services implemented by the container.*

These services include transactions, security, events, and persistence. The component uses this reference to invoke operations required by the implementation at runtime beyond what is specified in its deployment descriptor.

7.4.2 Interfaces Common to both Container Types

This section describes the interfaces and operations provided by both **container types** to support all categories of CORBA components.

7.4.2.1 The ComponentContext Interface

The **ComponentContext** is an **internal** interface which provides a component instance with access to the common container-provided runtime services applicable to both **container types**. It serves as a "bootstrap" to the various services the container provides for the component.

> *The* **ComponentContext** *is intended to be the analogue of* **EJBContext** *in Enterprise Java Beans.*

The **ComponentContext** provides the component access to the various services provided by the container. It enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };

local ComponentContext {
    CORBA::Object get_reference ()
        raises (IllegalState);
    HomeBase get_home();
    Transaction get_transaction();
    HomeRegistration get_home_registration ();
    Security get_security();
    Events get_events();
};
```

get_reference

The **get_reference** operation is used to obtain the reference used to invoke the component. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

get_home

The **get_home** operation is used to obtain a reference to the home interface. The home is the interface which supports factory and finder operations for the component and is defined by the **home** declaration in component IDL.

get_transaction

The **get_transaction** operation is used to access the **Transaction** interface. The **Transaction** interface is used to implement component-managed transactions.

get_home_registration

The get_home_registration operation is used to obtain a reference to the **HomeRegistration** interface. The **HomeRegistration** is used to register component homes so they may be located by the **HomeFinder**.

get_security

The **get_security** operation is used to access the **Security** interface. The **Security** interface is used to access the current CORBA security credentials or test them against a required set of credentials.

get_events

The **get_events** operation is used to obtain a reference to the **Events** interface. The **Events** interface is used by the component to emit or publish events for external consumption or to subscribe to events it needs to process.

> *EJB uses its context interface for both accessors to services and operations on those services. It also specializes the EJBContext for use by the Session-Bean (SessionContext) and the EntityBean (EntityContext). CORBA components defines the context interfaces based on* **container type** *and uses the accessed interfaces to support operations for transactions, security, events, and persistence.*

7.4.2.2 The Home Interface

A home is an **external** interface which supports factory and finder operations for the component. These operations are generated from the **home** IDL declaration (see Section 5.8 on page 62). The context supports an operation (**get_home**) to obtain a reference to the component's home interface.

7.4.2.3 The BaseOrigin Interface

The **BaseOrigin** is an **internal** interface used by the component to request passivation when the current operation completes. Each **container type** specializes the **BaseOrigin** to add additional operations.The lifetime of a origin is equal to the lifetime of the container that implements it. The derived context interfaces support an operation to obtain a reference to a **container type** specific origin interface.

```
exception PolicyMismatch { };

local BaseOrigin {
    void req_passivate ()
        raises (PolicyMismatch);
};
```

req_passivate

The **req_passivate** operation is used by the component to inform the container that it wishes to be passivated when its current operation completes. To be valid, the component must have a servant lifetime policy of **component** or **container**. If not the **PolicyMismatch** exception is raised.

> *The BaseOrigin interface and its derived interfaces have no analogue in EJB. This is because CORBA has persistent object references which Java does not. EJB uses the Handle to emulate a persistent reference much like COM uses Monikers.*

7.4.2.4 The Transaction Interface

A server component may be either **transaction-unaware** or **transaction-aware**. A transaction-unaware component implementation relies on the transaction policy declarations packaged with the deployment descriptor and contains no transaction APIs in its implementation code.

> *This is similar to the default processing of an Enterprise Java Bean or a MTS component.*

A transaction-aware component may use the CORBA transaction service directly to manipulate the current transaction or it may choose to use a simpler API, defined by this specification, which exposes only those transaction demarcation functions needed by the component implementation.

Manipulation of the current transaction must be consistent between the client, the transaction policy specified in the deployment descriptor, and the component implementation.

> *For example, if the client or the container starts a transaction, the component may not end it (***commit** *or* **rollback***). The rules to be used are defined by the CORBA transaction service.*

If the component uses the **CosTransactions::Current** interface, all operations defined for **Current** may be used as defined by the CORBA transaction service with the following exceptions:

- The **Control** object returned by **suspend** may only be used with **resume**.
- Operations on **Control** are not supported with CORBA components and may raise the **NO_IMPLEMENT** system exception.

> *The* **Control** *interface in the CORBA transaction services supports accessors to the* **Coordinator** *and* **Terminator** *interfaces. The* **Coordinator** *is used to build object versions of XA resource managers. The* **Terminator** *is used to allow a transaction to be ended by someone other than the originator. Since neither function is within the scope of the demarcation subset of CORBA transactions used with CORBA components, we allow CORBA transaction services implementations used with CORBA components to raise the* **NO_IMPLEMENT** *exception. This provides the same level of function as the* **bean-managed** *transaction policy in Enterprise Java Beans.*

The **Transaction** is an **internal** interface implemented by the container. Because the **Transaction** is a wrapper over **CosTransactions::Current**, it is thread specific. The **Transaction** exposes a simple demarcation subset of the CORBA transaction service to the component. The context supports an operation (**get_transaction**) to obtain a reference to the **Transaction** interface. The **Transaction** interface is defined by the following IDL:

```
exception NoTransaction { };
exception InvalidCookie{ };

enum Status {
        ACTIVE,
        MARKED_ROLLBACK,
        PREPARED,
        COMMITTED,
        ROLLED_BACK,
        NO_TRANSACTION,
        PREPARING,
        COMMITTING,
        ROLLING_BACK
};

local LocalCookie {
    boolean same_as (in LocalCookie cookie);
};

local Transaction {
    void begin ();
    void commit ()
        raises (NoTransaction);
    void rollback ()
        raises (NoTransaction);
    void set_rollback_only ()
        raises (NoTransaction);
    boolean get_rollback_only()
        raises (NoTransaction);
    LocalCookie suspend ()
        raises (NoTransaction);
    void resume (in LocalCookie cookie)
        raises (InvalidCookie);
    Status get_status();
    void set_timeout (in long to);
};
```

begin

The **begin** operation is used by a component to start a new transaction and associate it with the current thread. When **begin** is issued by a component, it results in a **CosTransaction::Current::begin** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

commit

The **commit** operation is used by a component to terminate an existing transaction normally. If no transaction is active, the **NoTransaction** exception is raised. When **commit** is issued by a component, it results in a **CosTransaction::Current::commit** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

rollback

The **rollback** operation is used by a component to terminate an existing transaction abnormally. If no transaction is active, the **NoTransaction** exception is raised. When **rollback** is issued by a component, it results in a **CosTransaction::Current::rollback** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

get_rollback_only

The **get_rollback_only** operation is used by a component to test if the current transaction has been marked for rollback. The **get_rollback_only** operation returns **TRUE** if the transaction has been marked for rollback, otherwise it returns **FALSE**. If no transaction is active, the **NoTransaction** exception is raised. When **get_rollback_only** is issued by a component, it results in a **CosTransaction::Current::get_status** being issued to the CORBA transaction service and the **status** value returned being tested for the **MARKED_ROLLBACK** state.

> *In EJB, this operation is defined on the EJBContext (***javax.ejb.EJBContext***) interface, rather than the UserTransaction (***javax.jts.UserTransaction***) interface. We have chose to define only accessor operations in the context.*

set_rollback_only

The **set_rollback_only** operation is used by a component to mark an existing transaction for abnormal termination. If no transaction is active, the **NoTransaction** exception is raised. When **set_rollback_only** is issued by a component, it results in a **CosTransaction::Current::rollback_only** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

> *In EJB, this operation is defined both on the EJBContext (***javax.ejb.EJBContext***) interface and the UserTransaction (***javax.jts.UserTransaction***) interface. We have chosen to define only accessor operations in the ComponentContext and thus a single* **set_rollback_only** *API.*

suspend

The **suspend** operation is used by a component to disconnect an existing transaction from the current thread. If no transaction is active, the **NoTransaction** exception is raised. The **suspend** operation returns a **cookie** which can only be used in a subsequent

resume operation. When **suspend** is issued by a component, it results in a **CosTransaction::Current::suspend** being issued to the CORBA transaction service. The rules for the use of this operation are more restrictive than the rules of its corresponding CORBA transaction service operation:

- Only one transaction may be suspended
- The suspended transaction is the only transaction that may be resumed.

resume

The **resume** operation is used by a component to reconnect a transaction previously suspended to the current thread. The **cookie** identifies the suspended transaction which is to be resumed. If the transaction identified by **cookie** has not been suspended, the **InvalidCookie** exception is raised. When **resume** is issued by a component, it results in a **CosTransaction::Current::resume** being issued to the CORBA transaction service. The rules for the use of this operation are more restrictive than the rules of its corresponding CORBA transaction service operation since the suspended transaction is the only transaction that may be resumed.

get_status

The **get_status** operation is used by a component to determine the status of the current transaction. If no transaction is active, it returns the **NoTransaction** status value. Otherwise it returns the state of the current transaction. When **get_status** is issued by a component, it results in a **CosTransaction::Current::get_status** being issued to the CORBA transaction service. The status values returned by this operation are equivalent to the status values of its corresponding CORBA transaction service operation.

set_timeout

The **set_timeout** operation is used by a component to associate a time-out value with the current transaction. The timeout value (**to**) is specified in seconds. When **set_timeout** is issued by a component, it results in a **CosTransaction::Current::set_timeout** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

> *The Transaction interface is equivalent to the UserTransaction interface (***javax.jts.UserTransaction***) in EJB with the addition of* **get_rollback_only**, **suspend** *and* **resume** *operations.*

7.4.2.5 The HomeRegistration Interface

The **HomeRegistration** is an **internal** interface which may be used by the CORBA component to register its home so it can be located by a **HomeFinder**.

> *The* **HomeRegistration** *interface allows a component implementation to advertise a home instance that can be used to satisfy a client's* **find_home_by_type** *request. It may also be used by an administrator to do the same thing. It is likely that the combination of* **HomeRegistration** *and*

HomeFinder *interfaces will work within the domain of a single container provider unless multiple implementations used other shareable directory mechanisms, e.g. an LDAP global directory. Federating* **HomeFinders** *is a similar problem to federating CORBA security domains and we defer to the security people for an architecture for such federation rather than attempting to specify such an architecture in this specification.*

The **HomeRegistration** interface is defined by the following IDL:

```
local HomeRegistration {
    void register_home (in HomeBase home);
    void unregister_home (in HomeBase home);
};
```

register_home

The **register_home** operation is used to register a component home with the **HomeFinder** so it can be located by a component clients. The **home** parameter identifies the home being registered.

unregister_home

The **unregister_home** operation is used to remove a component home from the **HomeFinder**. Once **unregister_home** completes, a client will never be returned a reference to the home specified as being unregistered. The **home** parameter identifies the home being unregistered.

7.4.2.6 The RemoteHomeRegistration Interface

The **RemoteHomeRegistration** is an **internal** interface, derived from **HomeRegistration**, which can be used by the CORBA component to register a remote home (i.e. one that is **NOT** collocated with the component) so it can be returned by a **HomeFinder**. The **RemoteHomeRegistration** interface is defined by the following IDL:

```
exception UnknownActualHome { };
exception RemoteHomeNotSupported { };

local RemoteHomeRegistration : HomeRegistration {
    void register_remote_home (
        in HomeBase rhome,
        in HomeBase ahome)
        raises (UnknownActualHome, RemoteHomeNotSupported);
};
```

register_remote_home

The **register_remote_home** operation is used to register a component home, not collocated with the instances that it can create, with the **HomeFinder** so it can be located by component clients. The **rhome** parameter identifies the home being

registered. If the **ahome** is not known, the **UnknownActualHome** exception is raised. If this component does not support remote homes, the **RemoteHomeNotSupported** exception is raised.

7.4.2.7 The Security Interface

Security on the server is primarily controlled by the security policy in the deployment descriptor for this component. The component may use CORBA security directly to determine the credentials associated with an operation or it may use the **Security** interface provided by the container. The context supports an operation (**get_security**) to obtain a reference to the **Security** interface. Because the **Security** is a wrapper over **SecurityLevel2::Current**, it is also thread specific. The **Security** interface is defined by the following IDL:

```
typedef SecurityLevel2::Credentials Principal;

local Security {
    Principal get_caller_identity();
    boolean is_caller_in_role (in Principal role);
};
```

get_caller_identity

The **get_caller_identity** operation obtains the CORBA security credentials in effect for the caller.

is_caller_in_role

The **is_caller_in_role** operation is used by the CORBA component to compare the current credentials to the credentials defined by the role parameter. If they match, **TRUE** is returned. If not, **FALSE** is returned.

> *This section of the EJB specification is in the process of being changed for EJB 1.1. We've defined a simple API for use by CORBA components, primarily to retrieve the credentials set by the client or the container using the deployment descriptor. We will need to harmonize this with EJB during the P-spec RTF.*
>
> *The Security APIs in EJB (which are the same as those defined above) are defined on the* **EJBContext** *interface (***javax.ejb.EJBContext***). We have chosen instead to define a distinct Security interface (like both EJB and CORBA components does for transactions) which has an accessor operation on the context.*

7.4.2.8 The Events Interface

The **Events** is an **external** interface which supports operations for emitting and publishing events and for subscribing to events emitted or published by others. These operations are generated from the **emits**, **publishes**, and **consumes** declaration in the component's IDL (see Section 5.7 on page 52). The context supports an operation (**get_events**) to obtain a reference to the **Events** interface.

EJB does not have an event API yet, but one is under development. The Java Programming Environment (JPE) does however have a messaging API (JMS) which supports publish/subscribe. This is another area that will need to be harmonized with EJB during the P-spec RTF.

7.4.2.9 The EnterpriseComponent Interface

All CORBA components must implement an interface derived from the **EnterpriseComponent** interface to be housed in a component container. **EnterpriseComponent** is a **callback** interface which defines no operations.

> **EnterpriseComponent** *is equivalent to the* **EnterpriseBean** *interface of Enterprise Java Beans.It supports operations to associate the context with the component.*

```
local EnterpriseComponent {
};
```

7.4.3 Interfaces Supported by the Transient Container Type

This section describes the interfaces supported by the transient **container type**. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this **container type**.

7.4.3.1 The TransientContext Interface

The **TransientContext** is an **internal** interface which provides a component instance with access to the container-provided runtime services. It serves as a "bootstrap" to the various services the container provides for the component.

> *The* **TransientContext** *is intended to be the analogue of* **SessionContext** *in Enterprise Java Beans.*

The **TransientContext** provides the component access to the various services provided by the container. It enables the component to simply obtain all the references it may require to implement its behavior.

```
local TransientContext : ComponentContext {
    TransientOrigin get_transient_origin();
};
```

> *EJB uses its context interfaces for both accessors to services and operations on those services. EJB specializes the EJBContext for use by the SessionBean (SessionContext). CORBA components defines the context interfaces based on* **container type** *and uses the accessed interfaces to support operations for transactions, security, events, and persistence.*

get_transient_origin

The **get_transient_origin** operation is used to obtain a reference to the **TransientOrigin** interface. The **TransientOrigin** interface is used to create and manage object references for the transient **container type**.

7.4.3.2 The TransientOrigin Interface

The **TransientOrigin** is an **internal** interface that extends the **BaseOrigin** interface by adding operations to create references for components deployed in a **transient container type**. It adds the ability to create references for these component categories. The **TransientOrigin** is defined by the following IDL:

```
local TransientOrigin : Origin {
    CORBA::Object create_ref (
        in CORBA::RepositoryId repid)
};
```

create_ref

The **create_ref** operation is used to create a reference to be exported to clients to invoke operations. The **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

7.4.3.3 The ServiceComponent Interface

The **ServiceComponent** interface is derived from the **EnterpriseComponent** interface and must be implemented by **service** components. **ServiceComponent** is a **callback** interface which provides a context to a **service** component to be used to access container services during its invocation.

> **ServiceComponent** *is similar to the* **SessionBean** *interface of Enterprise Java Beans but it is optimized for the* **service** *component which is always stateless. The EJB* **SessionBean** *may be stateless or stateful so we map it to the* **session** *component and not the* **service** *component.*

```
local ServiceComponent : EnterpriseComponent {
    void set_transient_context (in TransientContext ctx);
};
```

set_transient_context

The **set_transient_context** operation is used to set the **TransientContext** of the component. The container calls this operation after a component instance has been created. This operation is called outside the scope of an active transaction.

7.4.3.4 The SessionComponent Interface

The **SessionComponent** is a **callback** interface implemented by a **session** CORBA component. It provides operations for disassociating a context with the component and to manage servant lifetimes for a **session** component.

> *The* **SessionComponent** *is analogous to the* **SessionBean** *interface of Enterprise Java Beans.*

```
local SessionComponent : ServiceComponent {
    void activate();
    void passivate();
    void remove ();
};
```

activate

This operation is called by the container to notify a **session** component that it has been made active. The component instance should perform any initialization required prior to operation invocation.

passivate

This operation is called by the container to notify a **session** component that it has been made inactive. The component instance should release any resources it acquired at activation time.

remove

The **remove** operation is called by the container when the servant is about to be destroyed. It informs the component that it is about to be destroyed.

7.4.3.5 The Synchronization Interface

The **Synchronization** interface is a **callback** interface which may optionally be implemented by the **session** component. It permits the component to be notified of transaction boundaries by its container.

> *The* **Synchronization** *interface is the analogue of the* **SessionSynchronization** *interface in EJB.*

```
local Synchronization {
    void before_completion ();
    void after_completion (
        in boolean committed);
};
```

before_completion

The **before_completion** operation is called by the container just prior to the start of the two-phase commit protocol. The container implements the **CosTransactions::Synchronization** interface of the CORBA transaction service and invokes the **before_completion** operation on the component before starting its own processing.

after_completion

The **after_completion** operation is called by the container after the completion of the two-phase commit protocol. If the transaction has committed the **committed** value is set to **TRUE**. If the transaction has been rolled back, the **committed** value is set to **FALSE**. The container implements the **CosTransactions::Synchronization** interface of the CORBA transaction service and invokes the **after_completion** operation on the component after completing its own processing.

> *The EJB* **SessionSynchronization** *interface also has an* **after_begin** *operation which notifies the bean that a new transaction has begun. It is not obvious how this is to be used but it is obvious that such an operation would require a modification to the CORBA transaction service to implement. Consequently, we have not defined this operation for CORBA components. If subsequent analysis determines this to be of value, we will add it to CORBA components along with a change to the CORBA transaction service to support it.*

7.4.4 Interfaces Supported by the Persistent Container Type

This section describes the interfaces supported by the persistent **container type**. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this **container type**.

7.4.4.1 The PersistentContext Interface

The **PersistentContext** is an **internal** interface which provides a component instance with access to the container-provided runtime services. It serves as a “bootstrap” to the various services the container provides for the component.

> *The* **PersistentContext** *is intended to be the analogue of* **EntityContext** *in Enterprise Java Beans.*

The **PersistentContext** provides the component access to the various services provided by the container. It enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };

local PersistentContext : ComponentContext {
    ComponentId get_component_id ()
        raises (IllegalState);
    PersistentOrigin get_persistent_origin();
    Storage get_storage ();
};
```

get_component_id

The **get_component_id** operation is used to obtain a reference to the **ComponentId** interface. The **ComponentId** interface encapsulates a persistence identifier which can be used to access the component's persistence state. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

get_persistent_origin

The **get_persistent_origin** operation is used to obtain a reference to the **PersistentOrigin** interface. The **PersistentOrigin** interface is used to create and manage object references for the persistent **container type**.

get_storage

The **get_storage** operation is used to access the **Storage** interface. The **Storage** interface is used by the component to implement component-managed persistence.

> *EJB uses the EntityContext interface for both accessors to services and operations on those services. CORBA components restricts the context interface to accessors and uses the referenced interfaces to support operations for transactions, security, events, and persistence.*

7.4.4.2 The ComponentId Interface

The **ComponentId** interface is an **internal** interface provided by the persistent **container type** to locate a component's persistent state in a persistent store. A **ComponentId** is encapsulated in every object reference associated with a component (i.e. the component reference, any supported references, and all the provided references). The **ComponentId** interface supports the use of a CORBA persistence mechanism as well as any user-defined mechanisms (e.g. ODBC) for accessing persistent stores. The **ComponentId** interface is defined by the following IDL:

```
exception DuplicateSegment { };
exception NoSuchSegment { };

const StoreType USER=0;
const StoreType PSS=1;

union StoreId switch StoreType {
    case USER : ApplId aid;
    case PSS : Persistence::PersistentId pid;
};

struct SegmentDescr {
    long segment;
    StoreId sid;
};

typedef sequence<SegmentDescr> SegmentList;

local ComponentId {
    void add_segment (in long segment,
        in StoreId sid)
        raises (DuplicateSegment);
    void set_segment (in long segment)
        raises (NoSuchSegment);
    long get_segment ();
    StoreId get_store_id ();
    StoreId get_component_id ();
    StoreId get_store_id_for_segment (in long segment)
        raises (NoSuchSegment);
    SegmentList get_segment_list ();
};
```

add_segment

The **add_segment** operation associates a new entity in a persistent store (**sid**) with the segment identified as **segment**. Segments are defined in Chapter 6. If this **sid** is already associated with a different segment, the **DuplicateSegment** exception is raised.

set_segment

The **set_segment** operation is used to associate this **ComponentId** with a new segment identified by **segment**. If the segment does not exist, the **NoSuchSegment** exception is raised.

get_segment

The **get_segment** operation is used to retrieve the **segment** associated with this **ComponentId**.

get_store_id

The **get_store_id** operation is used to retrieve the persistence identifier associated with this **ComponentId**. For storage managed by CORBA persistence, this will be a **PersistentId**. For storage managed by other mechanisms, this will be the **AppId** used by the component to create the **ComponentId**.

get_component_id

The **get_component_id** operation is used to retrieve the persistence identifier associated with the component interface of this **ComponentId**. For storage managed by CORBA persistence, this will be a **PersistentId**. For storage managed by other mechanisms, this will be the **AppId** used by the component to create the **ComponentId**.

get_store_id_for_segment

The **get_store_id_for_segment** operation is used to retrieve the persistence identifier associated with a specific segment (identified by **segment**) of this **ComponentId**. For storage managed by CORBA persistence, this will be a **PersistentId**. For storage managed by other mechanisms, this will be the **AppId** used by the component to create the **ComponentId**. If the segment is not defined, the **NoSuchSegment** exception is returned.

get_segment_list

The **get_segment_list** operation returns a sequence of segments associated with this **ComponentId**.

7.4.4.3 The PersistentOrigin Interface

The **PersistentOrigin** is an **internal** interface that extends the **BaseOrigin** interface by adding operations for creating and obtaining object references. Object references for components deployed in a persistent **container type** can choose to use the CORBA persistence mechanism or some other user defined mechanism. The **ComponentId** interface (defined in Section 7.4.4.2 on page 167) encapsulates this distinction when a reference is to be used. Operations for creating references are defined for both persistence mechanisms. The **PersistentOrigin** is defined by the following IDL:

```
enum BadComponentReferenceReason {
    NON_LOCAL_REFERENCE,
    NON_COMPONENT_REFERENCE,
    WRONG_CONTAINER,
    NOT_CREATED_WITH_AID,
    NOT_CREATED_WITH_PID
};

exception BadComponentReference {
    BadComponentReferenceReason reason
};

typedef sequence<octet> ApplId
typedef CORBA::NVList Criteria;

local PersistentOrigin : BaseOrigin {
    ComponentId create_cid_from_aid (
        in ApplId aid);
    ComponentId create_cid_from_pid (
        in Persistence::PersistentId pid);
    HomeBase get_home_by_cid (
        in ComponentId cid);
    CORBA::Object create_ref_from_cid (
        in CORBA::RepositoryId repid,
        in ComponentId cid,
        in Criteria crit);
    ComponentId get_cid_from_ref (
        in CORBA::Object ref)
        raises (BadComponentReference);
    ApplId get_aid_from_cid (
        in ComponentId cid)
        raises (BadComponentReference);
    Persistence::PersitentId get_pid_from_cid (
        in ComponentId cid)
        raises (BadComponentReference);
};
```

create_cid_from_aid

The **create_cid_from_aid** operation is used by a persistent component to create a **ComponentId** for persistent state managed by some user-defined persistence mechanism. The **ApplId** value **aid** is a user-managed identifier for the location of the state data in some persistent store.

create_cid_from_pid

The **create_cid_from_pid** operation is used by a persistent component to create a **ComponentId** for persistent state managed by the CORBA persistence mechanism. The **PersistentId** value **pid** identifies the incarnation in some persistent store.

get_home_by_cid

The **get_home_by_cid** operation is used to locate the home interface for this **ComponentId**. The home interfaces supports factory and finder operations for this component. A home interface needs to be narrowed to a type specific home from the **HomeBase** reference returned by this operation.

create_ref_from_cid

The **create_ref_from_cid** operation is used by a component factory to create an object reference which can be exported to clients. The **cid** parameter specifies the **ComponentId** value to be placed in the object reference and made available (using the **get_component_id** operation on the context) when the **PersistentComponent callback** operations are invoked. The **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created. The **crit** parameter specifies user-supplied parameters which can be used in creating the reference.

> *The semantics of the* **crit** *parameter are determined by container-specific configuration data not defined in this specification. The parameter is supported portably in the sense that all references returned can be used by the client regardless of who created them. Individual implementations may interpret the* **crit** *parameter to place information in the object key that adds value to their implementation. An example of such a use would be the partitioning of a persistent store by key range and returning references to component servers that exploit that key range partitioning.*

get_cid_from_ref

The **get_cid_from_ref** operation is used by a persistent component to retrieve the **ComponentId** encapsulated in the reference (**ref**). The **ComponentId** interface supports operations to locate the state in some persistent store. The **BadComponentReference** exception can be raised if the input reference is not local (**NON_LOCAL_REFERENCE**), not a component reference (**NON_COMPONENT_REFERENCE**), or created by some other container (**WRONG_CONTAINER**).

get_pid_from_cid

The **get_pid_from_cid** operation is used by a component that uses CORBA persistence to manage its state to retrieve the **PersistentId** (**pid**) it provided when the **ComponentId** was created. The **PersistentId** provides a persistent handle to an incarnation in the persistent store.The **BadComponentReference** exception can be raised if the input **ComponentId** was not created from a **PersistentId** (**NOT_CREATED_WITH_PID**).

get_aid_from_cid

The **get_aid_from_cid** operation is used by a persistent component that manages its own state without using CORBA persistence to retrieve the **AppId** (**id**) it provided when the **ComponentId** was created. The **AppId** provides a persistent handle to an

application-defined location of the persistent state. The **BadComponentReference** exception can be raised if the input **ComponentId** was not created from a **ApplId** (**NOT_CREATED_WITH_AID**).

7.4.4.4 The Storage Interface

The **Storage** is an **internal** interface which can be used by the CORBA component to access operations related to the CORBA persistence service. The **Storage** interface is a wrapper on a CORBA persistence mechanism required by CORBA components.

> *The* **StorageHome** *interface defined in Chapter 6 of this specification identifies all the function required by CORBA components. If a CORBA persistence specification is ultimately adopted which supports all the operations defined in Chapter 6, the* **Storage** *interface can remain as defined below. If not, we will isolate ourselves from the details of the CORBA persistence service by adding the equivalent operation on* **StorageHome** *to the* **Storage** *interface which will have to map between these APIs and the CORBA persistence APIs.*

The **Storage** interface is defined by the following IDL:

```
exception InvalidCategory { };

local Storage {
    Persistence::StorageHomeBase get_storage_home (
        in Persistence::StorageHomeId homeid)
        raises (InvalidCategory,
            Persistence::HomeNotAvailable);
    PrimaryKey get_primary_key ()
        raises (InvalidCategory);
};
```

get_storage_home

The **get_storage_home** operation provides the component access to the persistence provider. It returns a **StorageHomeBase** interface which supports operations for managing incarnations in a persistent store. The **StorageHomeId** identifies the persistent store to be used. These interfaces are defined in Chapter 6. If the **get_storage_home** operation is issued by a **session** component, the **InvalidCategory** exception is raised.

get_primary_key

The **get_primary_key** operation is used by an **entity** component to access the primary key value declared for this component. This operation is equivalent to issuing the same operation on the component's **HomeBase** interface. If the **get_primary_key** operation is issued by a **session** or **process** component, the **InvalidCategory** exception is raised.

7.4.4.5 The PersistentComponent Interface

The **PersistentComponent** is a **callback** interface implemented by both **process** and **entity** components. It contains operations to manage the persistent state of the component.

> **PersistentComponent** *is equivalent to the* **EntityBean** *interface in Enterprise Java Beans.*

```
local PersistentComponent : EnterpriseComponent {
    void set_persistent_context (in PersistentContext ctx);
    void unset_persistent_context ();
    void activate ();
    void load ();
    void store ();
    void passivate ();
    void remove ();
};
```

set_persistent_context

The **set_persistent_context** operation is used to set the **PersistentContext** of the component. The container calls this operation after a component instance has been created. This operation is called outside the scope of an active transaction.

unset_persistent_context

The **unset_persistent_context** operation is used to remove the **PersistentContext** of the component. The container calls this operation just before a component instance is destroyed. This operation is called outside the scope of an active transaction.

activate

The **activate** operation is called by the container to notify the component that it has been made active. When container-managed persistence is implemented using CORBA persistence, this operation can be implemented in generated code. The component instance should perform any initialization required prior to operation invocation.

load

The **load** operation is called by the container to instruct the component to synchronize its state by loading it from its underlying persistent store. If CORBA persistence is being used, the component can extract its **PersistentId** from the **ComponentId** to locate its state in the persistent store. If CORBA persistence is not being used, the component can extract its **ApplId** from the **ComponentId** to locate its state in the persistent store. This operation executes within the scope of the active transaction.

store

The **store** operation is called by the container to instruct the component to synchronize it state by saving it in its underlying persistent store. If CORBA persistence is being used, the component can extract its **PersistentId** from the **ComponentId** to

determine where to save its state in the persistent store. If CORBA persistence is not being used, the component can extract its **AppId** from the **ComponentId** to determine where to save its state in the persistent store. This operation executes within the scope of the active transaction.

passivate

The **passivate** operation is called by the container to notify the component that it has been made inactive. When container-managed persistence is implemented using CORBA persistence, this operation can be implemented in generated code. The component instance should release any resources it acquired at activation time.

remove

The **remove** operation is called by the container when the servant is about to be destroyed. It informs the component that it is about to be destroyed. This operation is always called outside the scope of a transaction.

> *The* **PersistentComponent** *interface is equivalent to the* **EntityBean** *in Enterprise Java Beans. Container-managed persistence with CORBA persistence supports automatic code generation for activate and passivate. The component implementor can augment this with other data in the load and store methods. Since both* **process** *and* **entity** *components can have persistent state and container-managed persistence, the same callback interfaces can be used.*

7.5 The Client Programming Model

This section describes the architecture of the component programming model as seen by the client programmer. The client programming model as defined by the IDL extensions has been described previously (Chapter 5). This chapter focuses on the use of standard CORBA by the client who wishes to communicates with a CORBA component implemented in a **Component Server**.

> *This material serves the same purpose at the " **Enterprise JavaBeans to CORBA Mapping**" specification does for EJB. It enables a CORBA client who is not itself a CORBA component, to communicate with a CORBA component using standard CORBA.*

The client interacts with a CORBA component through two forms of external interfaces - a **home** interface and one or more **application** interfaces. Home interfaces support operations which allow the client to obtain references to an application interface which the component implements.

From the client's perspective, the home supports two design patterns - factories for creating new objects and finders for existing objects. These are distinguish by the presence of a **primaryKey** parameter in the home IDL.

- if a **primaryKey** is defined, the home supports both factories or finders and the client may use both.
- if a **primaryKey** is not defined, the home supports only the factory design pattern and the client must create new instances.

Two forms of clients are supported by the CORBA component model:

- Component-aware clients - These clients know they are making requests against a component (as opposed to an ordinary CORBA object) and can therefore avail themselves of unique component function, e.g. navigation among multiple interfaces and component type factories.
- Component-unaware clients - These clients do not know that the interface they are making requests against is implemented by a CORBA component so they can only invoke functions supported by an ordinary CORBA object, e.g. looking up a name in a Naming or Trader service, searching for a particular type of factory using a factory finder, etc.

7.5.1 Component-aware Clients

Clients that are defined using the IDL extensions in Chapter 5 are referred to as **component-aware** clients. Such clients can avail themselves of the unique features of CORBA components which are not supported by ordinary CORBA objects. The interaction between these clients and a CORBA component are outlined in the following sections. A **component-aware** client interact with a component through one or more CORBA interfaces:

- the interface associated with the **component** IDL declaration,
- zero or more supported interface declared on the **component** specification.
- zero or more interfaces defined by the **provides** clauses in the **component** definition,
- the home interface which supports factory and finder operations

Furthermore a component-aware client locates those interfaces using the **Components::HomeFinder** or a naming service. The starting point for client interactions with the component is the **resolve_initial_references** operation on **CORBA::ORB** which provides the initial set of object references.

7.5.1.1 Initial References

Initial references for all services used by a component client are obtained using the **CORBA::ORB::resolve_initial_references** operation. This operation currently supports the following references required by a component client:

- Name Service (**“NameService”**)
- Transaction Current (**“TransactionCurrent”**)
- Security Current (**“SecurityCurrent”**)
- Notification Service (**“NotificationService”**)
- Interface Repository (**“InterfaceRepository”**) for DII clients.

Additionally, this specification adds **Components::HomeFinder**. This reference is obtained using a new **ObjectID, "ComponentHomeFinder**" with **CORBA::ORB::resolve_initial_references**. The client uses this operation to obtain a reference to the **HomeFinder** interface. This requires the following enhancement to the **ORB** interface definition:

```
module CORBA {

        interface ORB {
        Object resolve_initial_references (in ObjectID identifier)
            raises (InvalidName);
        };
};
```

The string, **"ComponentHomeFinder"** is added to the list of valid **ObjectID** values.

7.5.1.2 Factory Design Pattern

For factory operations, the client invokes a **create** operation on the home. Default **create** operations are defined for each category of CORBA components for which code can be automatically generated. These operations return an object of type **CORBA::Component** which must be narrowed to the specific type. Alternatively, the component designer may specify custom factories as part of the **component** definition to define a type-specific signature for the **create** operation. Because these operations are defined in IDL, operation names can be chosen by the component designer. All that is required is that the operations return an object of the appropriate type.

A client using the factory design pattern uses the **HomeFinder** to locate the component factory (**HomeBase**) by interface type. The **HomeFinder** returns a type-specific factory reference which can then be used to create new instances of the component interface. Once created, the client makes operation requests on the reference representing the interface. This is illustrated by the following code fragment below:

```
// Resolve HomeFinder
org.omg.CORBA.Object objref =
orb.resolve_initial_references(“ComponentHomeFinder”);

ComponentHomeFinder ff =
ComponentHomeFinderHelper.narrow(objref);

org.omg.CORBA.Object of =
ff.find_home_by_type(AHomeHelper.id());

AHome F = AHomeHelper.narrow (of);
org.omg.Components.ComponentBase AInst = F.create();
A Areal = AHelper.narrow (AInst);

// Invoke Application Operation
answer = A.foo(input);
```

7.5.1.3 Finder Design Pattern

A component-aware client wishing to use an existing component instance (rather than create a new instance) uses a **finder** operation. Finders are supported for **entity** components only. Client’s may use the **HomeFinder** as described in Section 5.9 on page 73 to locate the component’s home or they may use CORBA naming to look up a specific instance of the home by symbolic name.

> *The latter choice is equivalent to the EJB model where the client uses **JNDI** (the Java version of CORBA naming) to look up* **EJBHome** *(which provides client interfaces to **factory** and **finder** services for Enterprise JavaBeans).*

A client using the finder design pattern uses the **CosNaming::NamingContext** interface to lookup a symbolic name. The naming service returns an object reference of the type previously bound. The client then makes operation requests on the reference representing the interface. This is illustrated by the following code fragment below:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references(“NamingService”);

NamingContext ncRef = NamingContextHelper.narrow(objref);

// Resolve the Object Reference in Naming
NameComponent nc = new NameComponent(“A“,””);
NameComponent path[] = {nc};
A aRef = AHelper.narrow(ncref.resolve(path));

// Invoke Application Operation
answer = A.foo(input);
```

7.5.1.4 Transactions

A component-aware client may optionally define the boundaries of the transaction to be used with CORBA components. If so, it uses the CORBA transaction service to ensure that the active transaction is associated with subsequent operations on the CORBA component.

The client obtains a reference to **CosTransactions::Current** by using the **CORBA::ORB::resolve_initial_references** operation specifying an **ObjectID** of **"TransactionCurrent"**. This permits the client to define the boundaries of the transaction, i.e. how many operations will be invoked within the scope of the client's transaction. All operations defined for **Current** may be used as defined by the CORBA transaction service with the following exceptions:

- The **Control** object returned by **get_control** and **suspend** may only be used with **resume**.
- Operations on **Control** may raise the **NO_IMPLEMENT** exception with CORBA components.

> *The* **Control** *interface in the CORBA transaction services supports accessors to the* **Coordinator** *and* **Terminator** *interfaces. The* **Coordinator** *is used to build object versions of XA resource managers. The* **Terminator** *is used to allow a transaction to be ended by someone other than the originator. Since neither function is within the scope of the demarcation subset of CORBA transactions used with CORBA components, we allow CORBA transaction services implementations used with CORBA components to raise the* **NO_IMPLEMENT** *exception.*

The following code fragment shows a typical usage:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references("TransactionCurrent");

Current txRef = CurrentHelper.narrow(objRef);
txRef.begin();
// Invoke Application Operation
answer = A.foo(input);
txRef.commit();
```

7.5.1.5 Security

A component-aware client uses the existing CORBA security mechanism to manage security for a CORBA component. There are two scenarios possible:

- Use of SSL for establishing client credentials

  CORBA security today does not define a standard API for clients to use with SSL to set the credentials which will be used to authorize subsequent requests. The credentials must be set in a way which is proprietary to the client ORB.

- Use of SECIOP by the client ORB.

In this case, CORBA security does define an API and it must be used by the client to establish the credentials to be used to authorize subsequent requests.

Security processing for CORBA components uses a subset of CORBA security. For SECIOP, the client sets the credentials to be used with subsequent operations on the component by using operations on the **SecurityLevel2::PrincipalAuthenticator**. The client obtains a reference to **SecurityLevel2::Current** by using the **CORBA::ORB::resolve_initial_references** operation specifying an **ObjectID** of **“SecurityCurrent”**. This permits the client to access the **PrincipalAuthenticator** interface to associate security credentials with subsequent operations. The following code fragment shows a typical usage:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references(“SecurityCurrent”);

org.omg.SecurityLevel2.PrincipalAuthenticator secRef =
org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow
(objRef);

secRef.authenticate(...);

// Invoke Application Operation
answer = A.foo(input);
```

7.5.1.6 Events

Component-aware clients wishing to **emit** or **consume** events use the component APIs defined in Chapter 5. Alternatively, they may use the CORBA Notification service APIs directly and conform to the subset supported by CORBA components (see Section 7.3.8 on page 146 for details).

7.5.2 Component-unaware Clients

CORBA components can also be used by clients who are unaware that they are making requests against a component. Such clients can see only a single interface (the supported interface of a component) and do not support navigation.

7.5.2.1 Initial References

Component-unaware clients obtain initial references using existing CORBA mechanisms, viz. **CORBA::ORB::resolve_initial_references**. It is unlikely, however, that this mechanism would be used to obtain a reference to the **HomeFinder**.

7.5.2.2 Factory Design Pattern

The factory design pattern can be used by component-unaware clients only if the supported interface has application operations defined. This permits existing CORBA objects to be easily converted to CORBA components, transparently to their existing clients. The following techniques can be used:

- The reference to a factory finder (typically the **CosLifeCycle::FactoryFinder**) can be stored in the Naming or Trader service and looked up by the client before creating the instance.
- A reference to the home interface can be obtained from the Naming service.

  *This technique is equivalent to the EJB client programming model which uses **JNDI** to look up a reference to* **EJBHome** *by name.*

- The reference to the home interface can be obtained from a Trader service.
- After locating a factory finder, the factory can be located using the existing **find_factories** operation or by using the new **find_factory** operation on the **CosLifeCycle::FactoryFinder** interface. The **find_factory** is defined in Chapter 14.

  The current **CosLifeCycle** ***find_factories*** *operation returns a sequence of factories to the client requiring the client to choose the one which will create the instance. To allow the server (i.e. the FactoryFinder) to make the selection, we also add a new* **find_factory** *operation to* **CosLifeCycle** *which allows the server to choose the "best" factory for the client request based on its knowledge of workload, etc.*

  A factory finder will return an **Object**. *A component-unaware client may expect to narrow this to* **CosLifeCycle::GenericFactory** *and use the generic* **create** *operation. This will fail unless the component implementor has specialized* **HomeBase** *and mixed in and implemented the* **GenericFactory** *interface.*

- A stringified object reference can be retrieved from a file known by the component-unaware client.

Once a reference to the home has been obtained, the client can create component instances and make operation requests on the component. Each component exports at least one IDL interface. The supported interface must be used by the client to invoke the component's application operations. Provided interfaces cannot be located using the factory design pattern.

7.5.2.3 Finder Design Pattern

A component-unaware client can use the CORBA Naming service to locate an existing **entity** component. Unlike the factory design pattern, the name to be looked up by the client can be either the supported interface or any of the provided interfaces. The following techniques can be used:

- A symbolic name associated with the component's home can be looked up in a Naming service to make an invocation of the **finder** operations.

*This technique is equivalent to the EJB client programming model which uses **JNDI** to look up a reference to* **EJBHome** *by name.*

- Alternatively, the reference to the home interface can be obtained from a Trader service.
- the **finder** operation can be invoked on the **entity** component to return a reference to the client.

7.5.2.4 Transactions

This is the same as component-aware clients (See Section 7.5.1.4 on page 178). However, the possibility of the **NO_IMPLEMENT** exception being raised for operations on **Control** may have a more serious impact, since the component-unaware client may not be expecting that to happen.

7.5.2.5 Security

This is the same as component-aware clients (See Section 7.5.1.5 on page 178).

7.5.2.6 Events

Component-unaware clients wishing to **emit** or **consume** events must use the equivalent CORBA Notification services interfaces and stay within the subset supported by CORBA components (see Section 7.3.8 on page 146 for details). This is illustrated by the following code fragment:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references(“NotificationService”);

org.omg.CosNotififyChannelAdmin.EventChannelFactory evfRef =
org.omg.EventChannelFactoryHelper.narrow(objRef);

// Create an Event Channel
org.omg.CosNotifyChannelAdmin.EventChannel evcRef =
evfRef.create_channel(...);

// Obtain a SupplierAdmin
org.omg.CosNotifyChannelAdmin.SupplierAdmin publisher =
evcRef.new_for_suppliers (...);

// And a ConsumerProxy
org.omg.CosNotifyComm.ProxyConsumer proxy =
publisher.obtain_notification_push_comsumer (...);

// Publish a structured event
proxy.push_structured_event(...);
```

Container Architecture 8

This chapter describes the architecture of the **component container** as seen by the container provider. The container is a server-side framework built on the ORB, the Portable Object Adaptor (POA), and a set of CORBA services which provides the runtime environment for a CORBA component. A component container may be implemented by an existing ORB vendor or by companies not in that business today using the facilities of a CORBA_3 ORB enhanced to support the core changes identified in this specification (see Section 14.1 on page 387 for details).

The container architecture in this chapter is described in terms of a specific design for building the container on the POA using a **ServantLocator**. Other design are also possible although there are specific combinations of POA policies that cannot be made to work. These are indicated in the text as rationale in the body of the chapter. A container implementation that exhibits the same behavior as the one presented in this chapter is conformant, even if it implements the container differently.

Unless otherwise noted, all interfaces in this chapter are defined within the **Container** module embedded within the **Components** module (see appendix A.1 on page 399 for the proposed naming structure for CORBA 3.0 suggested by this specification).

8.1 Change History

The following changes have been made since the December 1998 version of the document (orbos/98-12-02) was posted:

1. Persistence has been factored into the container architecture.
2. The **ContainerFactory** interface has been introduced to explain the initial creation of component containers.
3. The Empty Container has been introduced to allow the component machinery to be used without any constraints on the existing CORBA_3 architecture.

4. The container creation mechanism has been made extensible to allow other specialized containers to be defined in the future. This includes partitioning the container space into OMG-defined and vendor-defined containers.

5. Since the **process** and **entity** containers use the same **ServantLocator**, the **ServantLocator** is no longer isomorphic with the container category so they have been renamed to **TransientServantLocator** and **PersistentServantLocator** which more accurately captures their essential differences.

6. Interface names have been synchronized with those defined elsewhere in the document and **Cookies** have replaced **Tokens** for consistency.

7. Miscellaneous clarifications have been made to the text.

The following changes have been made since the February version of the document (orbos/98-02-01):

1. The **ServantLocator** designed presented in this chapter has been made an example rather than the only way to implement the container.

2. There are now two component categories which use **TRANSIENT** references. This requires a new POA to manage the servant to oid ratio properly, but it can use the same **ServantLocator**.

3. The **executor** has been incorporated in place of the servant where appropriate.

4. Operations for creating containers and installing components have been removed and are now specified in Chapter 9. As a result, none of the IDL defined in this chapter is normative.

5. Terminology has been updated to be consistent with the rest of the document

6. Additional details on the container requirements for interacting with the persistence mechanism including serializations and caching requirements have been added.

7. Threading requirements and the containers responsibility have been documented.

8. Miscellaneous clarifications have been made to the text.

All changes are clearly marked with change bars. In general existing text which was moved will not have change bars.

8.2 Component Server

A **Component Server** is a process which includes an arbitrary number of **Component Containers**. Each **container type** has an associated **container implementation type** which describes its interaction with the POA and the ORB. Each **container type** is capable of managing a specific set of **component categories**. Each **container type** includes a specialized POA[1] managed by a **Container Manager** which is responsible for creating and destroying the containers based on descriptive information packaged

1.The term “POA” is used to refer to not only the interface **Poa**, but all the related interfaces (**ServantManager**, **ServantLocator**, etc.) necessary to create references and activate object instances in response to client requests.

with the component. **Container Managers** are factories which support the creation of **Component Containers** and are created as part of the installation and deployment process for CORBA components. The details of deployment are described in Section 9.9 on page 282. The overall architecture is depicted in Figure 8-1 below:

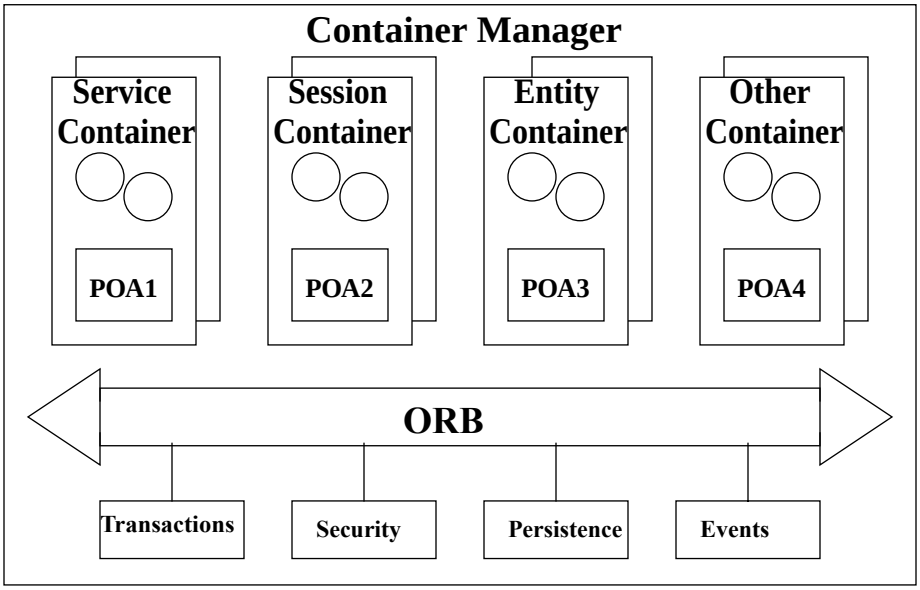

Figure 8-1 A Component Server

A component container is created as a result of component deployment as outlined in Chapter 9. The container specification is translated by the Container Manager into a set of POA policies, a container API framework, and a set of CORBA service bindings that will be used by the container. This enables the container to implement **internal** interfaces which offer services to the component and invoke **callback** interfaces which the component developer must implement.

8.2.1 POA Creation

A POA is used to create references that will be exported to clients and to handle activation of component instances when operation requests are received. Creating a container usually involves the creation of a POA[2] which the container will use. The **container implementation type** associated with a particular **container type** determines some of the policies which must be associated with the POA. These have been previously described in Section 7.3 on page 140. Others which are orthogonal to the container functionality (e.g. the use of firewall proxies) can be passed as input to **create_container**. It is the responsibility of the **create_container** operation to then create a POA which satisfies these requirements.

2.It may be possible in some cases to actually use the root POA. This is not excluded, but has not been validated.

CORBA::ORB::resolve_initial_references with an **ObjectID** of **"RootPOA"** is used to locate the root POA. The **component category** determines the **container implementation type** to be created. The container factory uses this information to create a POA and its associated interfaces and to bind the container API framework associated with the **container type**.

> *The* **create_container** *design for creating a POA described below uses a* **ServantLocator** *architecture which enables specialized* **ServantManager** *interfaces to implement the container function by being on the invocation path for all requests directed to the component. The API frameworks and their associated deployment descriptors defined for the* **container types** *in this specification require the container to intervene before and after each operation request to implement the required function. This precludes certain POA policy choices, e.g. the use of a* **ServantActivator** *which is only called when the requested object is not in the POA's active object map. While other designs using different POA policies may be possible, this one was chosen because it best describes how the container behavior needs to be implemented.*

The steps required are as follows:

- The **CORBA::Policy** objects required by the POA are created with the proper values. The **container implementation type** requires or (in some cases) suggests specific POA policies. An example of a set that will work for each **container implementation type** can be found in Section 8.3 on page 189.
- A POA is created using the **POA:create_poa** operation specifying a sequence of the **Policy** objects created in the previous step as input. The complete set of **Policy** objects includes the mandatory set (dictated by the **container implementation type**), the orthogonal set (specified as input to container creation), and the implementation-specific set (chosen by the container provider to deliver the proper semantics).
- The **container type** value is used to determine which **ServantManager** should be assigned to the POA (**POA::set_servant_manager**).
  - For the **fwork-a container type**, the **ServantManager** is set to the **TransientServantLocator** (see Section 8.6.1 on page 223).
  - For the **fwork-b container type**, the **ServantManager** is set to the **PersistentServantLocator** (see Section 8.6.2 on page 225).
- The newly created POA is then activated (**POA::activate**)

The container implementation is actually provided by the **ServantManager** interface. A **ServantLocator** is used to allow the container to be on the invocation path for every operation request. These POAs specify the **USE_SERVANT_MANAGER** policy, enabling a **ServantManager** to be used to associate a servant with the request to instantiate the object. Normally the **ServantManager** interface is implemented by user applications, but in this design, it is implemented by the container provider.

To install a CORBA component in a container a handshake is required between the component and the container to exchange references between the component implementation and the container.

8.2.2 Binding the Container to CORBA services

The **container type** identifies which CORBA services will be used by the container. The **container types** defined in this specification use the following CORBA services:

- security
- transactions
- persistence
- notification
- naming

As part of container creation, accessibility to these CORBA services must be established and bindings created. At a minimum, this includes the use of the **resolve_initial_references** operation on **CORBA::ORB** to obtain initial references to these services. It also includes processing any container specific configuration data required for a particular service, e.g.

- setting up the channels to be used for emitting and consuming events,
- creating and initializing database connections to be used for persistence, and
- determining the naming context to be used to resolve component local names.

8.2.3 Container API Frameworks

The **container types** defined by this specification provide **frameworks** into which a CORBA component is deployed. We define two **container types** and their associated APIs in this specification. The framework manages interactions with the ORB, the POA, and the CORBA services on behalf of the CORBA component, allowing the component developer to concentrate on application logic. The major functions handled by the API frameworks (in association with the ORB, POA, and the CORBA services) include:

- creating object references
- factories and finders
- transactions
- security
- events
- persistence

A brief description of each of these is provided in the following sections.

8.2.3.1 Creating Object References

In CORBA, object references are created and managed by the POA. A component container creates these reference with specialized information which comes from either the container provider, the component implementor, or the persistence provider, depending on both the **component category** and the deployment options specified.

8.2.3.2 Factories and Finders

Factory and finder operations are declared using the **home** IDL declaration and are associated with the component's home interface. The container provides access to this interface at runtime and supports a set of operations for externalizing component homes for use by external clients.

8.2.3.3 Transactions

The container interacts with the CORBA transaction service on behalf of the component. Transaction policies, defined in the deployment descriptor, are translated into CORBA transaction service operations. The container also provides the **Transaction** interface, a simplified form of the demarcation part of the CORBA transaction service which the component implementor uses to support transaction functions at runtime.

8.2.3.4 Security

The container interacts with the CORBA security service on behalf of the component. Security policies, defined in the deployment descriptor, are translated into CORBA security service operations. The container also provides the **Security** interface, a simplified form of the application part of CORBA security which the component implementor uses to support security functions at runtime.

8.2.3.5 Events

The container provider is responsible for setting up and managing the event channels used by CORBA notification to support the component event model. The component event model relies on configuration information, local to the container implementor, to handle quality of service properties, filters, and the number and types of event channels.The container also provides access to the **Events** interface, which is defined by the component IDL specification, to allow the component to both generate and process events. Integrating the component event model with CORBA notification is addressed in Section 8.5 on page 221.

8.2.3.6 Persistence

Persistence is supported with the **fwork-b container type**. The **fwork-a container type** does not support persistence. The container also provides the **Storage** interface, a set of APIs which provides the functions required of CORBA persistence for the component implementor to use to implement component-managed persistence. Persistence considerations are covered in more detail in Section 8.4 on page 218.

8.2.3.7 Threading

CORBA components support two forms of thread safety: **serialize**, and **multithread**. These choices are described in Section 7.3.2.3 on page 142. The container implements these choices by either ensuring that only a single thread enter a component at a time (**serialize**) or by allowing multiple threads to enter a component simultaneously (**multithread**).

8.3 Containers Categories

This specification defines **container categories** corresponding to the four **component categories** with their associated framework APIs and an empty container to support creation of user-defined frameworks:

- The **Empty** container makes the entire suite of interfaces defined for CORBA_3 available to a component's implementation without restriction.
- The **Service** container which manages the **service** component designed for high-performance access to stateless CORBA components.
- The **Session** container which manages the **session** component for stateful CORBA components with transient state.
- The **Process** container which manages stateful **process** components which encapsulates all data access in the server using any persistence mechanism.
- The **Entity** container which manages stateful **entity** components which shares data access responsibility between the client and the server using any persistence mechanism.

The container categories are one to one with their **component categories**. The relationship between **component categories**, **container types** and **container implementation types** was described previously in Section 7.2 on page 137. The following sections describe each of the container categories in more detail.

8.3.1 The Empty Container

The **Empty** container exposes all CORBA functions directly to the component developer. No framework is provided to simplify programming, however all the tools necessary to build such a framework are available to the component developer. The component developer can choose any function currently defined in CORBA.The empty container is the means by which the advanced functions of CORBA components (e.g.

multiple interfaces, packaging, and deployment) are made available to any CORBA applications, including those that do not fit the profiles of the other containers.This is illustrated in Figure 8-2 below:

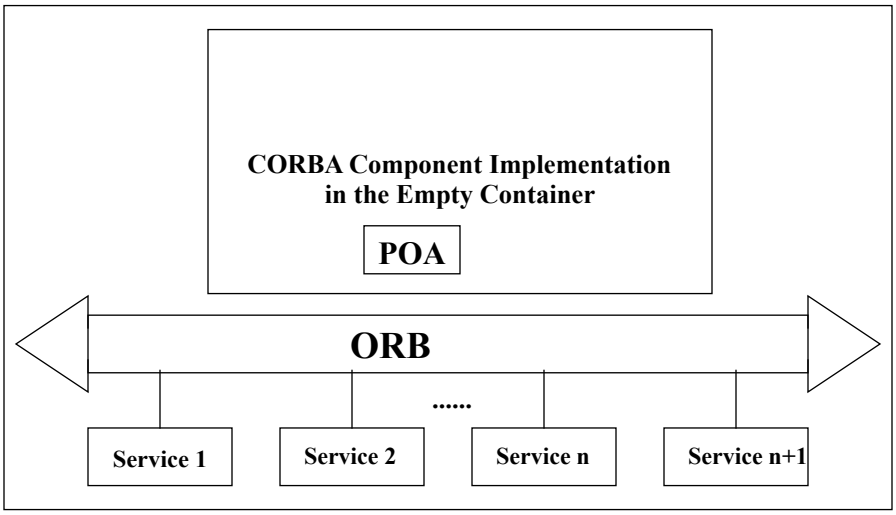

Figure 8-2 The Empty Container

Note that a CORBA component deployed in the empty container can use any arbitrary set (including the null set) of CORBA services. This specification places no constraints on what can be used within the existing CORBA architecture.

8.3.2 The Service Container

The **service** container implements the runtime environment for a **service** component. A **service** container can be implemented using a POA with the following policies:

Table 8-1 POA Policies for a Service Container

Policy Name	Required Value
Thread	ORB_CTRL_MODEL
Lifespan	TRANSIENT
ObjectId Uniqueness	MULTIPLE_ID
ID Assignment	SYSTEM_ID
Implicit Activation	NO_IMPLICIT_ACTIVATION
Servant Retention	NO_RETAIN
Transaction Policy	ALLOWS_SHARED

Table 8-1 POA Policies for a Service Container

Policy Name	Required Value
Request Processing	USE_SERVANT_MANAGER
Servant Manager	**TransientServantLocator**

Thread

The choice of **ORB_CTRL__MODEL** allows the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

Since **service** components have neither state nor identity, the use of **TRANSIENT** object references is the appropriate choice.

ObjectId uniqueness

A policy of **MULTIPLE_ID** allows the **service** container to assign any servant to any **ObjectId**. Since **service** components are stateless, any servant is capable of supporting any **ObjectId**

ObjectId assignment

A policy of **SYSTEM_ID** allows the POA to assign **ObjectId** values. Since **service** components have no identity, the **service** container has no need to manage **ObjectId** assignment.

implicit activation

This policy has no relevance to component containers hence it is set to **NO_IMPLICIT_ACTIVATION**.

servant retention

A policy of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A policy of **ALLOWS_SHARED** permits the container to set transaction policy based on the component's deployment descriptor.

request processing

The choice of **USE_SERVANT_MANAGER** allows the container to be implemented in the **ServantManager**.

8.3.2.1 Creating Object References

For **service** components, **ObjectIds** have no meaning since a **service** component has neither state or identity.

8.3.2.2 Factories and Instances

A component home implementation for a **service** component creates object references and component instances in response to the client's **create** requests. A **service** component's implementation registers its home with the **HomeFinder** to make it available to clients through find operations. For **service** components, the component instance and its home need not be collocated. Since instances have no state, they can be created anywhere when a request is received. The **HomeFinder** can also be located anywhere since it is a righteous CORBA object. Object references for both the component's supported interfaces and any provided interface are created by the POA within the **service** container.

8.3.2.3 *Invoking an Operation*

Figure 8-3 below outlines the steps necessary to make an operation invocation on a **service** component:

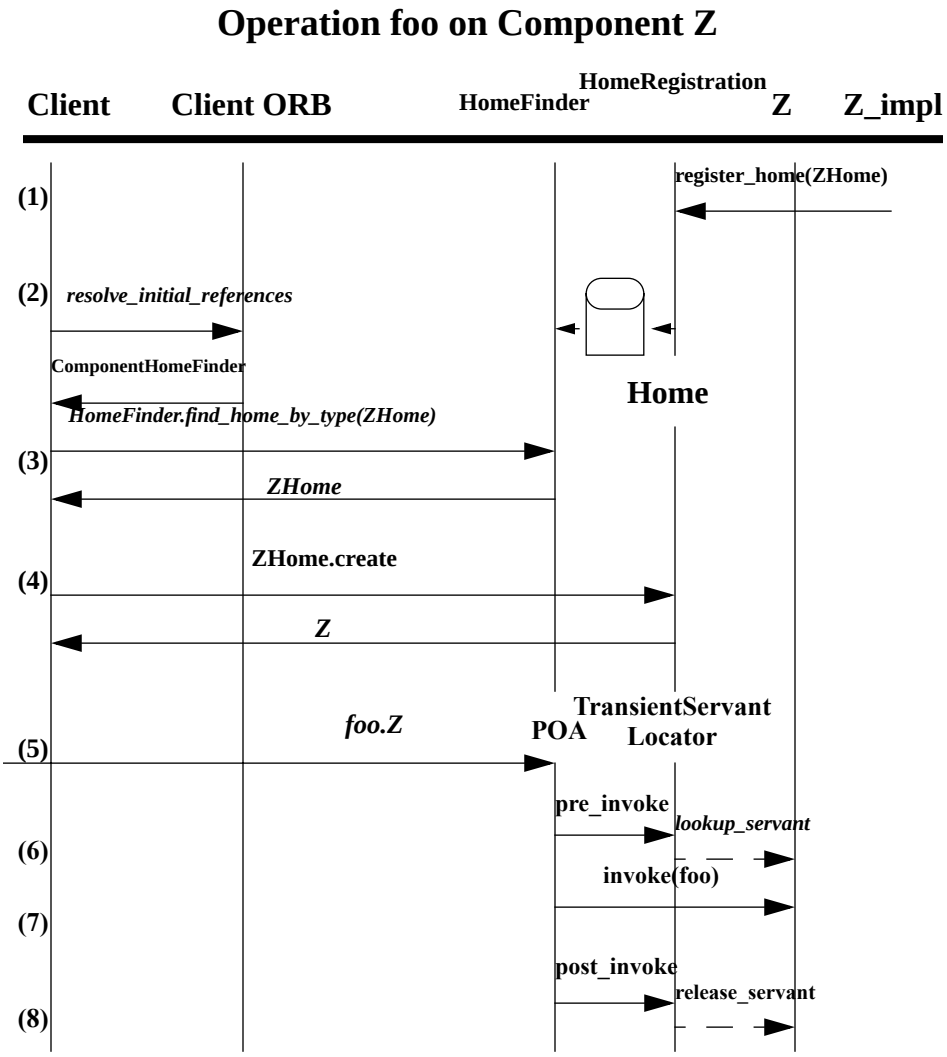

Figure 8-3 Using a Service Component

1. Component implementation registers a **service** component factory with the **HomeFinder** (**HomeRegistration.register_home**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.
3. Client uses the **HomeFinder.find_home_by_type** operation to find a factory (**Zhome**) that creates component instances of type **Z**.

4. Client invokes a **create** operation on the factory (**ZHome.create**). Since **Z** is a **service** component, the factory creates a reference and defers activation.
5. Client invokes the **foo** operation on **Z** (**Z.foo**).
6. The POA invokes the **TransientServantLocator** and requests an **executor** to process the request (**TransientServantLocator.pre_invoke**).The **TransientServantLocator** locates an appropriate **executor** or creates a new one using the **ExecutorFactory**. It returns the associated servant to the POA.
7. The POA dispatches the request to the component implementation (**Invoke Z.foo**).
8. After the request completes, the POA invokes the **TransientServantLocator (TransientServantLocator.post_invoke)**. The **TransientServantLocator** releases the associated **executor** to the pool.

8.3.2.4 Servant Lifetime Management

The **service** container supports a servant lifetime policy of **method**. A servant with a **method** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface and passivated in the **post_invoke** following the operation invocation. This behavior is shown in Figure 8-4 below:

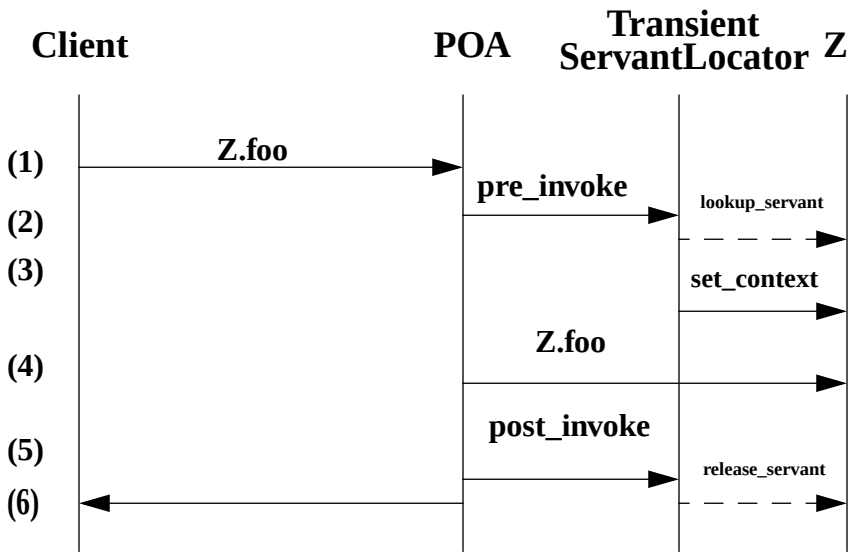

Figure 8-4 Service Container with a Method Lifetime Policy

1. Client invokes **foo** operation on **Z** (**Z.foo**).
2. POA invokes **pre_invoke** operation on **ServantManager** (**TransientServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.

4. POA then dispatches **foo** operation to **Z**.
5. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**).
6. POA then returns **foo** response back to client Since the servant lifetime policy is **method**, the **executor** is released.

8.3.3 The Session Container

The **session** container implements the runtime environment for a **session** component. A **session** container can be implemented using a POA with the following policies:

Table 8-2 POA Policies for a Service Container

Policy Name	Required Value
Thread	ORB_CTRL_MODEL
Lifespan	TRANSIENT
ObjectId Uniqueness	UNIQUE_ID
ID Assignment	USER_ID
Implicit Activation	NO_IMPLICIT_ACTIVATION
Servant Retention	NO_RETAIN
Transaction Policy	ALLOWS_SHARED
Request Processing	USE_SERVANT_MANAGER
Servant Manager	**TransientServantLocator**

Thread

The choice of **ORB_CTRL__MODEL** allows the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

Since **session** components have transient state and identity, the use of **TRANSIENT** object references is the appropriate choice.

ObjectId uniqueness

Since **session** components have identity, a policy of **UNIQUE_ID** is required to allow the container to distinguish between multiple instances of the same type.

ObjectId assignment

The **session** container will assign unique **ObjectIds** without input from the component implementation. This supports a structuring of **ObjectId** values which the container can exploit within its implementation.

implicit activation

This policy has no relevance to component containers hence it is set to **NO_IMPLICIT_ACTIVATION**.

servant retention

A policy of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A policy of **ALLOWS_SHARED** permits the container to set transaction policy based on the component's deployment descriptor.

request processing

The choice of **USE_SERVANT_MANAGER** allows the container to be implemented in the **ServantManager**.

8.3.3.1 Creating Object References

For **session** components, **ObjectIds** are managed by the **session** container without involvement from the component implementor. The container implementor is responsible for maintaining uniqueness. This permits **ObjectIds** to be encapsulated by the container provider in implementation specific ways.

8.3.3.2 Factories and Instances

The home implementation for a **session** component creates object references and component instances in response to the client's **create** requests. A **session** component's implementation registers its home with the **HomeFinder** to make it available to clients through find operations. For **session** components, the component instance and the factory must be collocated. The **HomeFinder**, however, can be located anywhere since it is a righteous CORBA object. Object references for both the component's supported interfaces and any provided interface are created by the POA within the **session** container.

8.3.3.3 *Invoking an Operation*

Figure 8-3 below outlines the steps necessary to make an operation invocation on a **session** component:

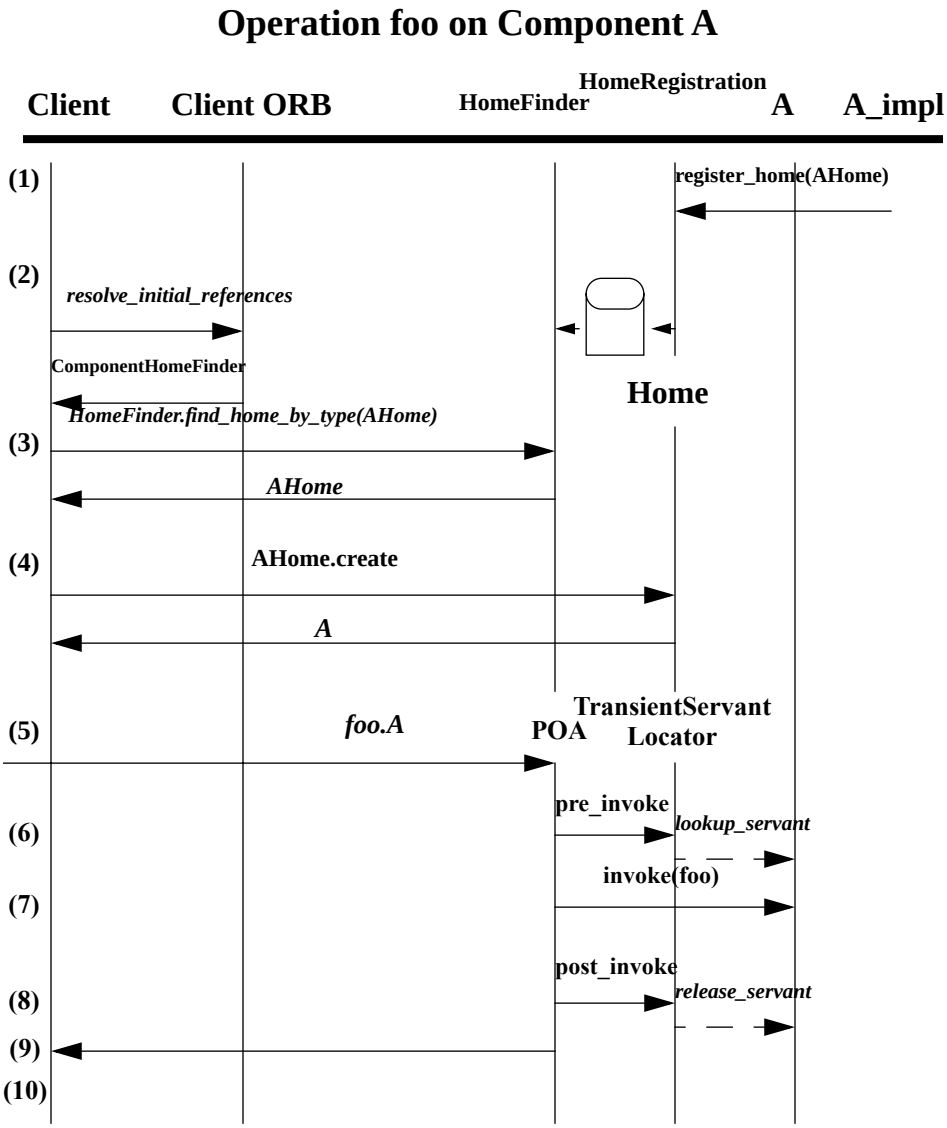

Figure 8-5 Using a Session Component

1. Component implementation registers a **session** component's home with the **HomeFinder** (**HomeRegistration.register_home**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.

3. Client uses the **HomeFinder.find_home_by_type** operation to find a factory (**Ahome**) that creates component instances of type **A**.
4. Client invokes a **create** operation on the factory (**AHome.create**). Since **A** is a **session** component, the factory creates a reference and may defer activation until the first operation invocation.
5. Client invokes the **foo** operation on **A** (**A.foo**).
6. The POA invokes the **TransientServantLocator** and requests an **executor** to process the request (**TransientServantLocator.pre_invoke**).The **TransientServantLocator** locates an appropriate **executor** or creates a new one using the **ExecutorFactory**. It returns the associated servant to the POA.
7. The POA dispatches the request to the component implementation (**Invoke A.foo**).
8. After the request completes, the POA invokes the **TransientServantLocator (TransientServantLocator.post_invoke)**.
9. POA then returns **foo** response back to client.
10. Steps [5] through [9] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **TransientServantLocator** releases the associated **executor** to the pool.

8.3.3.4 Servant Lifetime Management

The **session** container supports multiple servant lifetime policy values. An executor is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface and is passivated in the **post_invoke** following the expiration of the servant lifetime policy. This is illustrated in the following sections:

Method Lifetime

A **session** component with a **method** lifetime policy has its **executor** activated on every **pre_invoke** and passivated on every **post_invoke**. This behavior is shown in Figure 8-6:

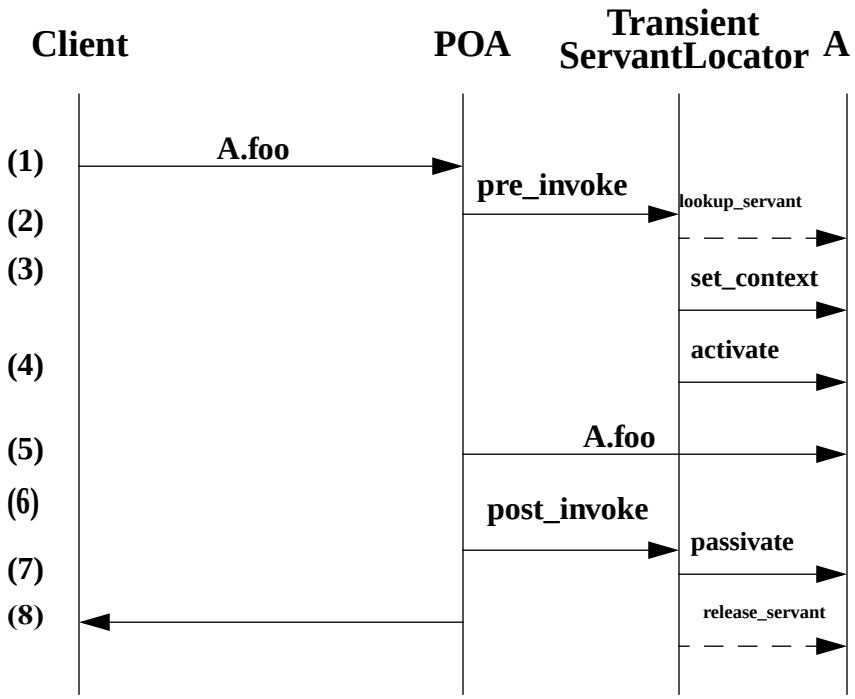

Figure 8-6 Session component with a Method Lifetime Policy

1. Client invokes **foo** operation on **A** (**A.foo**).
2. POA invokes **pre_invoke** operation on **ServantManager** (**TransientServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.
4. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
5. POA then dispatches **foo** operation to **A**.
6. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**).
7. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement this operation.
8. POA then returns **foo** response back to client and releases **executor**.

Transaction Lifetime

A **session** component with a **transaction** lifetime policy is activated on the first **pre_invoke** within a new transaction. Subsequent **pre_invoke** operations do not cause activation. Passivation occurs when the current transaction completes (successfully or unsuccessfully). The **TransientServantLocator** implements this policy using the CORBA transaction service **CosTransactions::Synchronization** interface. This behavior is shown in Figure 8-7:

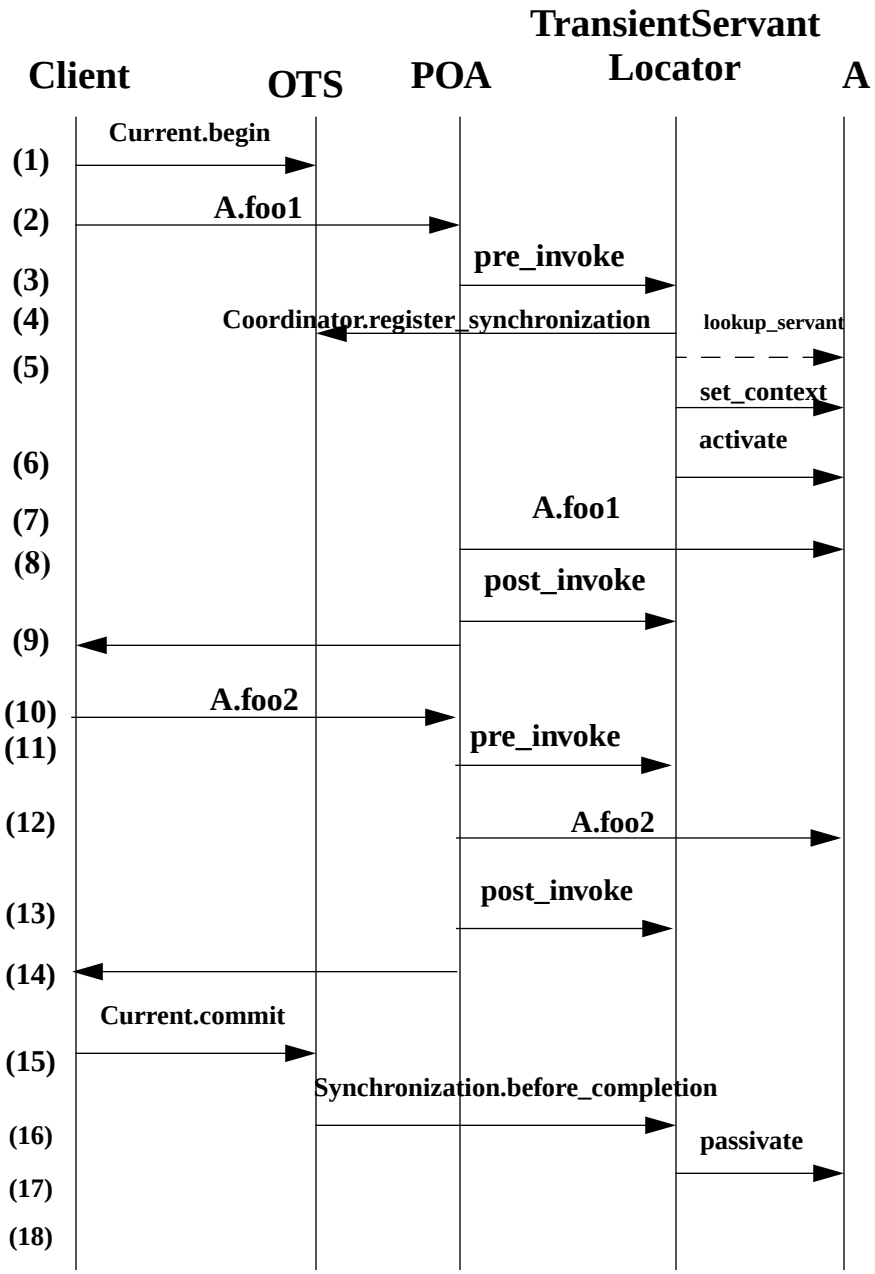

Figure 8-7 Session Component with a Transaction Lifetime Policy

1. Client begins a transaction with the CORBA transaction service (**Current.begin**)
2. Client invokes **foo1** operation on **A (A.foo1**).
3. POA invokes **pre_invoke** operation on **ServantLocator** (**TransientServantLocator.pre_invoke**).
4. **ServantLocator** registers a **Synchronization** object with the CORBA transaction service (**Coordinator.register_synchronization**) to be called by the CORBA transaction service at the start of the commit process.
5. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.
6. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
7. POA then dispatches **foo1** operation to **A**.
8. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**).
9. POA then returns **foo1** response back to client.
10. Client invokes **foo2** operation on **A**.
11. POA invokes **pre_invoke** operation on **ServantLocator** (**TransientServantLocator.pre_invoke**). Since **A** is already active, the **ServantLocator** returns to the POA.
12. POA then dispatches **foo2** operation to **A**.
13. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**).
14. POA then returns **foo2** response back to client.
15. Client attempts to terminate the transaction by calling commit (**Current.commit**)
16. CORBA transaction service notifies **ServantLocator** prior to the start of phase one of commit (**Synchronization.before_completion**).
17. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement this operation.
18. CORBA transaction service continues the two-phase commit process.

Component Lifetime

A **session** component with a **component** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application requested passivation or when the process terminates, whichever occurs first. This behavior is shown in Figure 8-8 below.

Container Lifetime

A **session** component with a **container** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application-requested passivation or in the **post_invoke** following an operation when the system needs to reclaim the memory, whichever occurs first. This behavior is identical to process behavior, except that failures can be simulated when the container determines that it needs to reclaim the memory associated with this component making it more likely that the final response will be returned to the client. This behavior is captured in Figure 8-8 below.

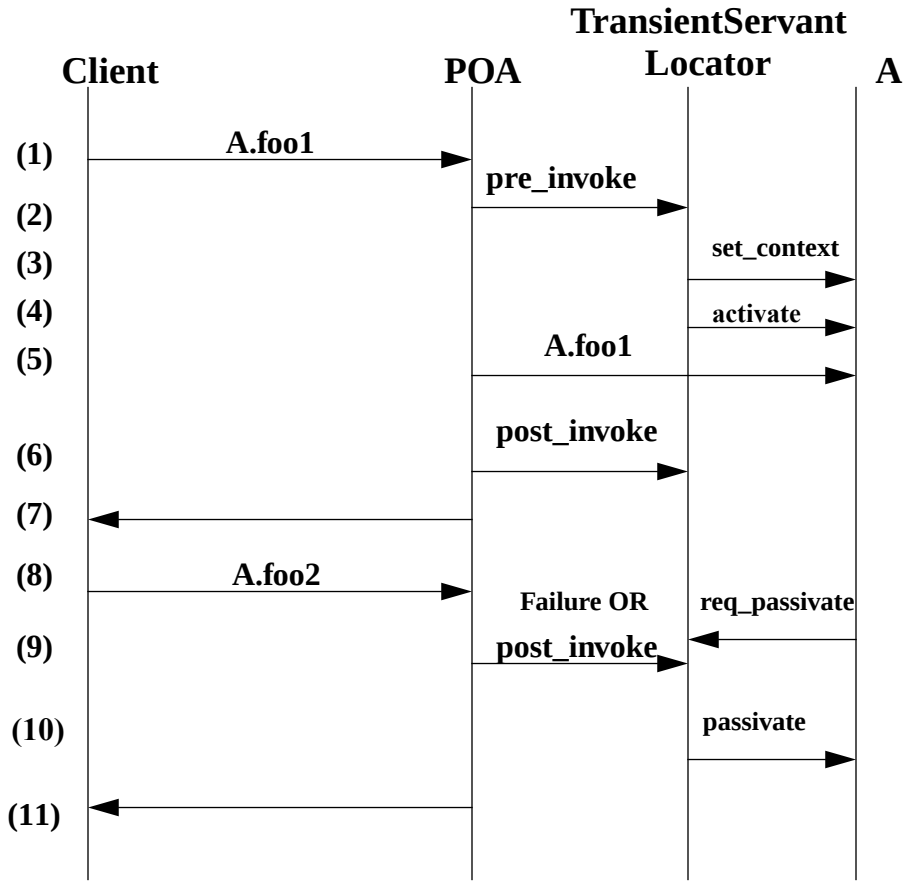

Figure 8-8 A Session Component with Component or Container Lifetime Policy

1. Client invokes **foo1** operation on **A** (**A.foo1**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**TransientServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.

4. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
5. POA then dispatches **foo1** operation to **A**.
6. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**). Since activation policy is **component** or **container**, the **ServantLocator** just returns to the POA.
7. POA then returns **foo1** response back to client.
8. Client continues invoking **foo2** operation (**A.foo2**). Either a failure occurs or **A** requests to be passivated (**Origin.req_passivate**).
9. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**TransientServantLocator.post_invoke**).
10. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement this operation.
11. POA then returns **foo2** response back to client (if possible).

8.3.4 The Process Container

The **process** container implements the runtime environment for a **process** component. A **process** container can be implemented using a POA with the following policies:

Table 8-3 POA Policies for a Process Container

Policy Name	Required Value
Thread	ORB_CTRL_MODEL
Lifespan	PERSISTENT
ObjectId Uniqueness	UNIQUE_ID
ID Assignment	USER_ID
Implicit Activation	NO_IMPLICIT_ACTIVATION
Servant Retention	NO_RETAIN
Transaction Policy	ALLOWS_SHARED
Request Processing	USE_SERVANT_MANAGER
Servant Manager	**PersistentServantLocator**

Thread

The choice of **ORB_CTRL__MODEL** allows the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

Since **process** components have both state and identity, the use of **PERSISTENT** object references is the appropriate choice.

ObjectId uniqueness

A policy of **UNIQUE_ID** allows the **process** container to distinguish between multiple equivalent instances.

ObjectId assignment

The **process** container will assign unique **ObjectIds** with input from the component implementation and the persistence mechanism. This not only supports a structuring of **ObjectId** values which the container can exploit within its implementation, but also makes it possible for the component implementor or the persistence mechanism to locate state from the **ObjectId**.

implicit activation

This policy has no relevance to component containers hence it is set to **NO_IMPLICIT_ACTIVATION**.

servant retention

A policy of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A policy of **ALLOWS_SHARED** permits the container to set transaction policy based on the component's deployment descriptor.

request processing

The choice of **USE_SERVANT_MANAGER** allows the container to be implemented in the **ServantManager**.

8.3.4.1 Creating Object References

The **process** container is responsible for creating and managing unique **ObjectIds** which can be used to locate an external copy of the component's persistent state. That state can be explicitly declared and managed by the container (**container-managed persistence**) or not declared and managed by the application (**component-managed persistence**). These **ObjectIds** are opaque both to the client and to the container, and may or may not use the CORBA persistence mechanism. This attribute makes it

possible to have factories for **process** components which create only object references and defer instance creation until an operation request is actually received. This enables workload to be distributed among several functionally equivalent servers.

8.3.4.2 Factories and Instances

The **process** component's home is responsible for creating references and exporting them to clients. Component instances are created on demand when a reference is used to invoke an operation.

Factory operations are typically invoked by clients but may also be invoked as part of the implementation of a specific interface provided by the component. A CORBA component implementation locates its home (which supports the factory operations) using the context provided by its container. Object references for both the component's interfaces and any provided interface are created by the POA which supports the container for that component.

8.3.4.3 Invoking an Operation

Figure 8-9 outlines the steps necessary to make an operation request on a **process** component:

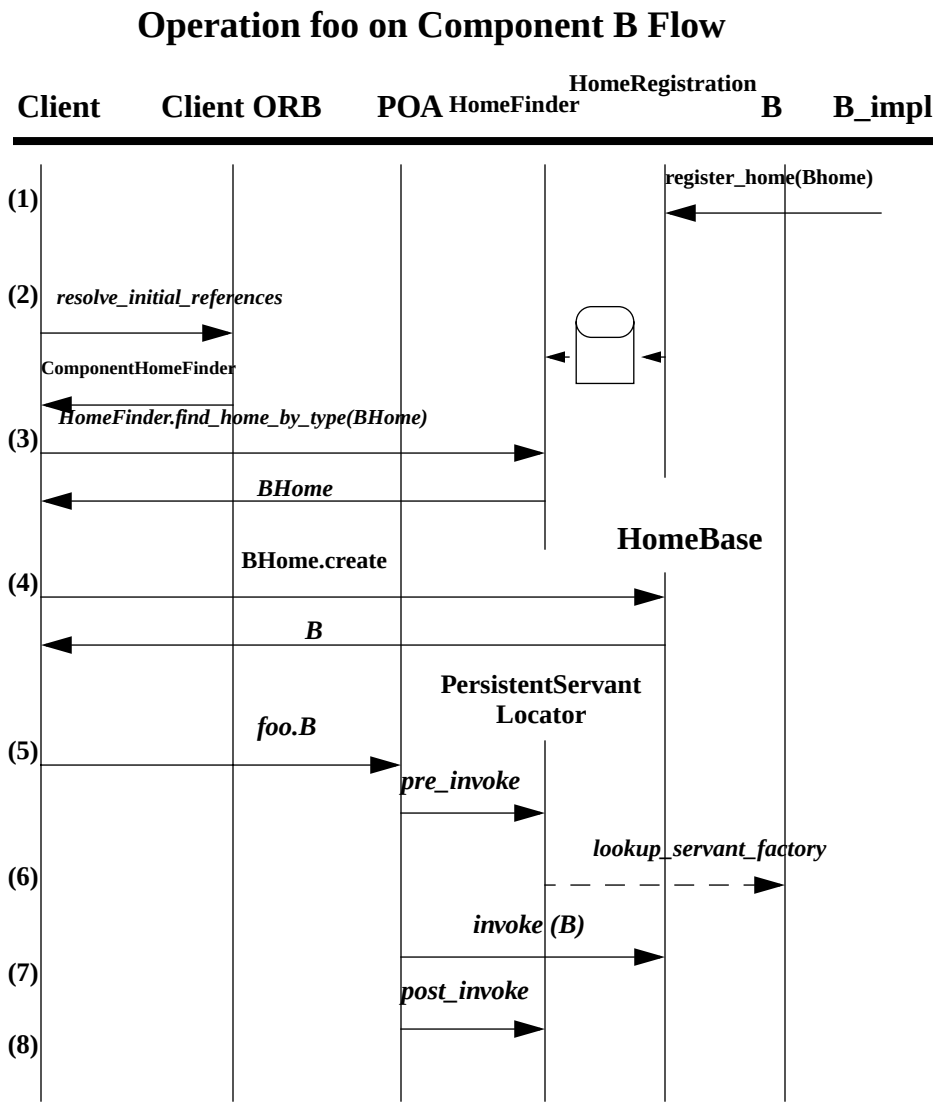

Figure 8-9 Using the Process Container

1. Component implementation registers a **process** component home with the **HomeFinder** (**HomeRegistration.register_factory**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.
3. Client uses the **HomeFinder.find_home_by_type** operation to find a factory (**BHome**) that creates component instances of type **B**.

4. Client invokes a **create** operation on the factory (**BHome.create**). Since **B** is **process** component, the factory need only create a reference; instance creation can be deferred until an operation is requested.

5. Client invokes the **foo** operation on **B** (**B.foo**). Since **B** is not active, the POA invokes the **pre_invoke** operation on the appropriate **ServantManager** (**PersistentServantLocator.pre_invoke**).

6. The **PersistentServantLocator** locates the **ExecutorFactory** and creates a new **executor** to handle the request. It then returns the associated servant to the POA to process the request.

7. The POA then dispatches the request to the servant (**invoke(B)**)

8. After the request completes, the POA invokes the **PersistentServantLocator (PersistentServantLocator.post_invoke)**.

9. Steps [5] through [8] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **PersistentServantLocator** releases the associated **executor** to the pool.

8.3.4.4 Servant Lifetime Management

The **process** component can have multiple servant lifetime policies specified in its deployment descriptor. The **PersistentServantLocator** implements these different policies by making activation decisions during **pre_invoke** and passivation decisions during **post_invoke**. This is illustrated in the following sections:

Method Lifetime

A **process** component with a **method** lifetime policy has its **executor** activated on every **pre_invoke** and passivated on every **post_invoke**. This behavior is shown in Figure 8-10:

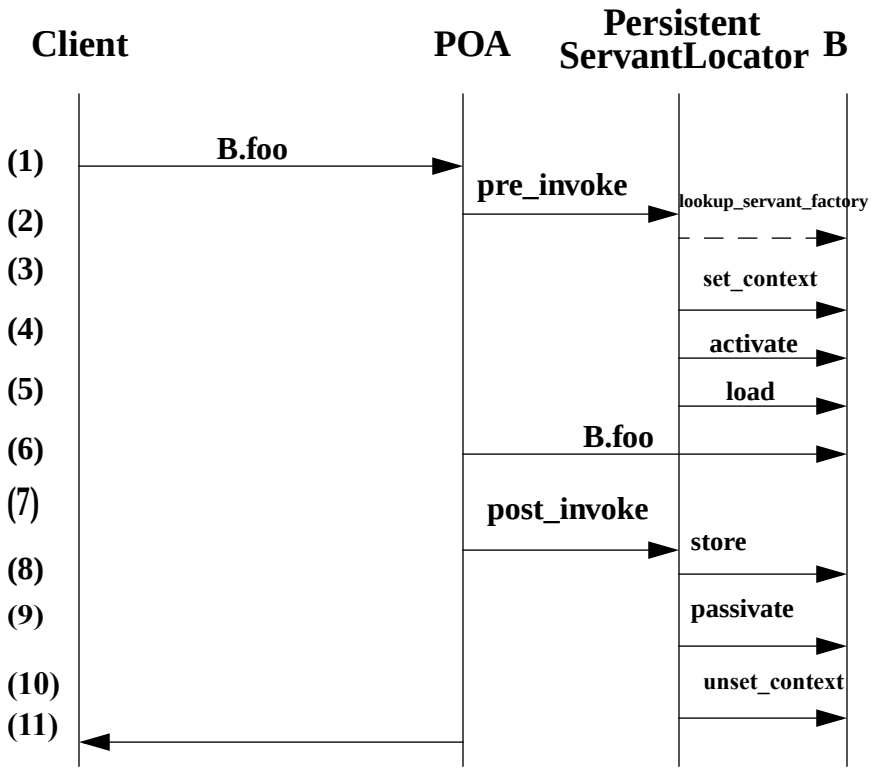

Figure 8-10 A Process Component with a Method Lifetime Policy

1. Client invokes **foo** operation on **B** (**B.foo**).
2. POA invokes **pre_invoke** operation on **ServantManager** (**PersistentServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.
4. **ServantLocator** creates a new **B** and invokes **activate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **activate** operation.
5. **ServantLocator** then invokes **load** callback operation. This enables **B** to locate its persistent state which is not declared and retrieve it from external storage.
6. POA then dispatches **foo** operation to **B**.

7. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**PersistentServantLocator.post_invoke**).

8. **ServantLocator** then invokes **store** callback operation. This enables **B** to save its persistent state, not explicitly declared, in some external storage location.

9. **ServantLocator** then invokes **passivate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement this operation.

10. **ServantLocator** invokes **unset_context** callback operation and releases the **executor**.

11. POA then returns **foo** response back to client.

Transaction Lifetime

A **process** component with a **transaction** lifetime policy has its **executor** activated on the first **pre_invoke** within a new transaction. Subsequent **pre_invoke** operations do not cause activation. Passivation occurs when the current transaction completes

(successfully or unsuccessfully). The **PersistentServantLocator** implements this policy using the CORBA transaction service **CosTransactions::Synchronization** interface. This behavior is shown in Figure 8-11:

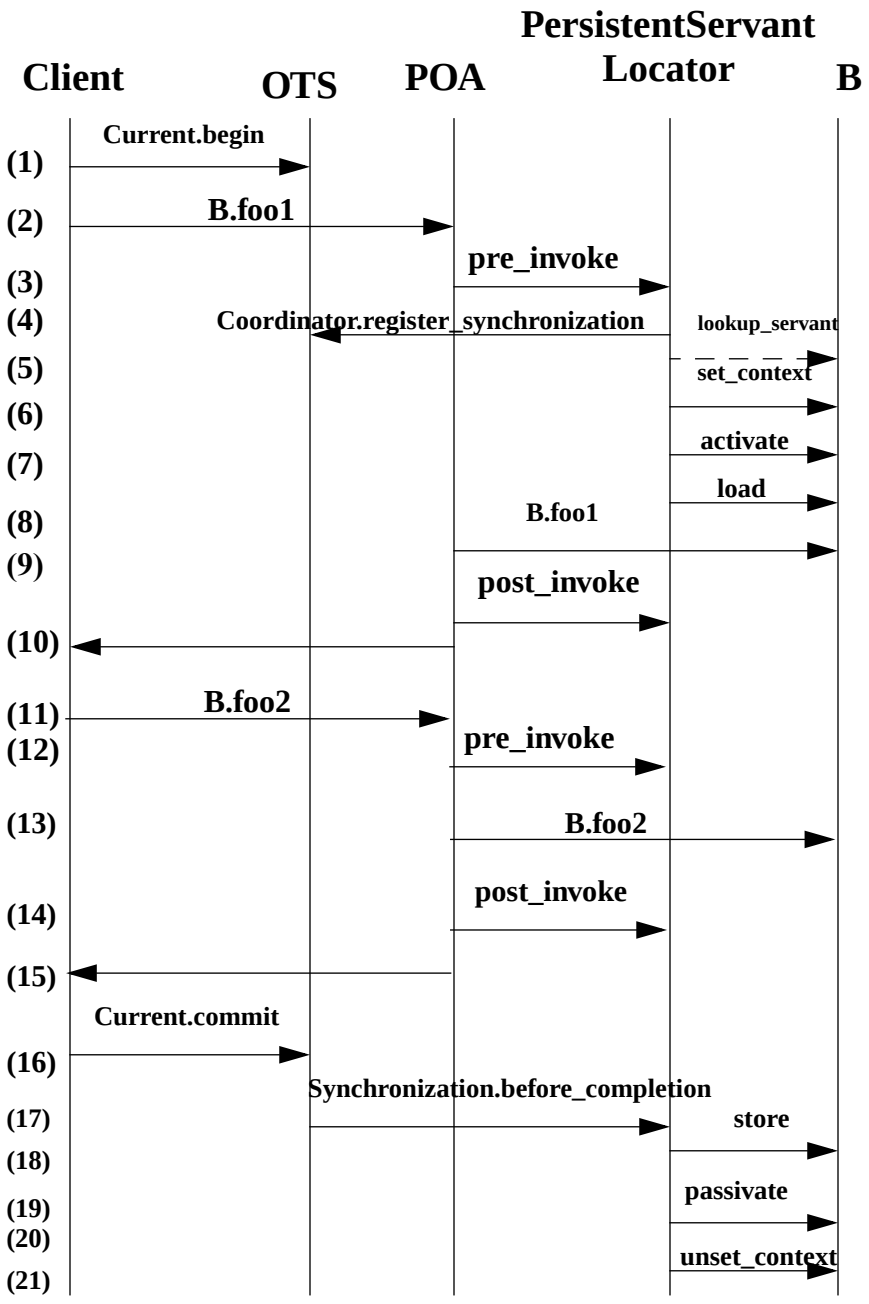

Figure 8-11 A Process Component with a Transaction Lifetime Policy

1. Client begins a transaction with the CORBA transaction service (**Current.begin**)
2. Client invokes **foo1** operation on **B (B.foo1**).

3. POA invokes **pre_invoke** operation on **ServantLocator** (**PersistentServantLocator.pre_invoke**).
4. **ServantLocator** registers a **Synchronization** object with the CORBA transaction service (**Coordinator.register_synchronization**) to be called by the CORBA transaction service at the start of the commit process.
5. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.
6. **ServantLocator** creates a new **B** and invokes **activate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **activate** operation.
7. **ServantLocator** then invokes **load** callback operation.This enables **B** to locate it's persistent state and retrieve it from external storage.
8. POA then dispatches **foo1** operation to **B**.
9. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**PersistentServantLocator.post_invoke**).
10. POA then returns **foo1** response back to client.
11. Client invokes **foo2** operation on **B**.
12. POA invokes **pre_invoke** operation on **ServantLocator** (**PersistentServantLocator.pre_invoke**). Since **B** is already active, the **ServantLocator** returns to the POA.
13. POA then dispatches **foo2** operation to **B**.
14. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**PersistentServantLocator.post_invoke**).
15. POA then returns **foo2** response back to client.
16. Client attempts to terminate the transaction by calling commit (**Current.commit**)
17. CORBA transaction service notifies **ServantLocator** prior to the start of phase one of commit (**Synchronization.before_completion**).
18. **ServantLocator** then invokes **store** callback operation. This enables **B** to save its persistent state in some external storage location.
19. **ServantLocator** then invokes **passivate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement this operation.
20. **ServantLocator** invokes **unset_context** callback operation and releases the **executor**.

21. CORBA transaction service continues the two-phase commit process.

Component Lifetime

A **process** component with a **component** lifetime policy has its **executor** activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application requested passivation or when the process terminates, whichever occurs first. This behavior is shown in Figure 8-12 below.

Container Lifetime

A **process** component with a **container** lifetime policy has its **executor** activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application-requested passivation or in the **post_invoke** following an operation when the system needs to reclaim the memory, whichever occurs first. This behavior is identical to process behavior, except that failures can be simulated when the container determines that it

needs to reclaim the memory associated with this component making it more likely that the final response will be returned to the client. This behavior is captured in Figure 8-12 below.

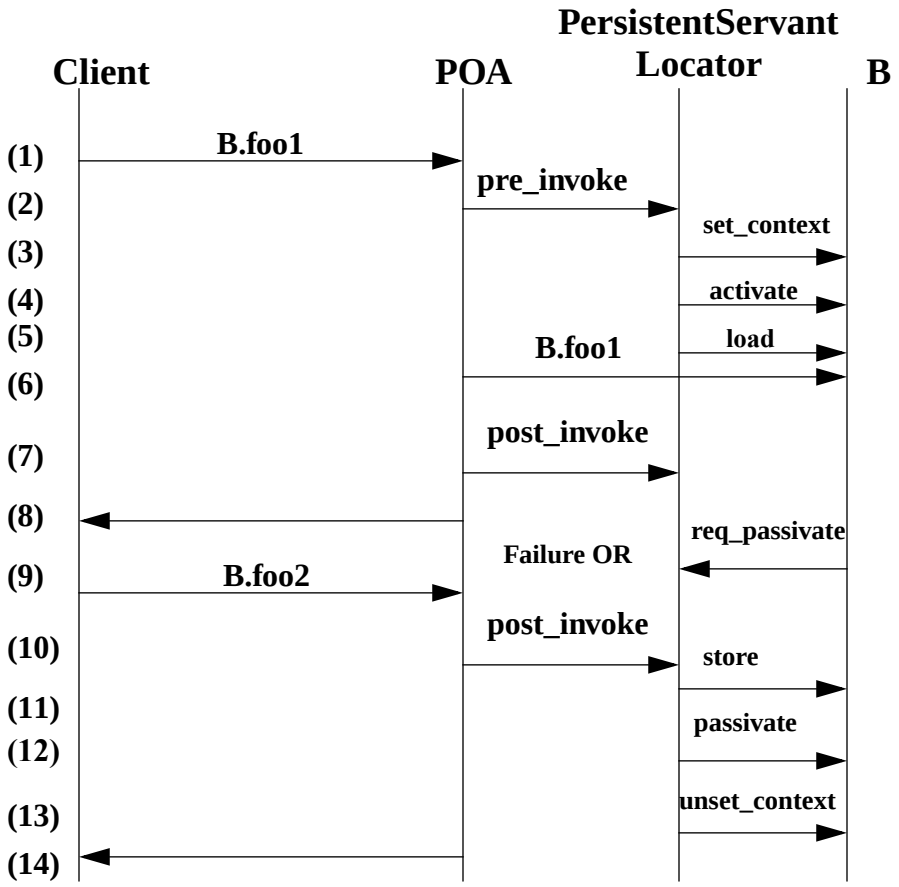

Figure 8-12 Process Component with Component or Container Lifetime Policies

1. Client invokes **foo1** operation on **B** (**B.foo1**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**PersistentServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes **set_context** callback operation.
4. **ServantLocator** invokes **activate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **activate** operation.
5. **ServantLocator** then invokes **load** callback operation.This enables **B** to locate it's persistent state and retrieve it from external storage.

6. POA then dispatches **foo1** operation to **B**.
7. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**PersistentServantLocator.post_invoke**). Since activation policy is **process**, the **ServantLocator** just returns to the POA.
8. POA then returns **foo1** response back to client.
9. Client invokes **foo2** operation on B (**B.foo2**). Either a failure occurs or **B** requests to be passivated (**Origin.req_passivate**).
10. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**PersistentServantLocator.post_invoke**).
11. **ServantLocator** then invokes **store** callback operation. This enables **B** to save its persistent state in some external storage location.
12. **ServantLocator** then invokes **passivate** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement this operation.
13. **ServantLocator** invokes **unset_context** callback operation and releases the **executor**.
14. POA then returns **foo** response back to client (if possible).

8.3.5 The Entity Container

The **entity** container provides the runtime environment for the **entity** component. The **entity** container can be implemented using the same POA policies as the **process** container. These were described in Table 8-3 on page 203.

8.3.5.1 Creating Object References

The **entity** container supports operations for associating primary keys with a **PersistentId** (**pid**). Every **entity** component instance is associated with one and only one primary key. The **entity** container provides operations on its **PersistentServantLocator** to create an **ObjectId** from a **pid**.

8.3.5.2 Factories and New Instances

A **entity** component's home is responsible for both creating references and creating new instances of **entity** components. Since **entity** components are also incarnations in a persistent store, creating a new instance of the **entity** component has the effect of creating a new record in a persistent store.

Factory operations are typically invoked by clients but may also be invoked as part of the implementation of a specific interface provided by the component. The **entity** component implementation locates its home (which supports the factory operations)

using the context provided by its container. Object references for both the component's interfaces and any provided interface are created by the POA which supports the container for the **entity** component.

8.3.5.3 Invoking an Operation on a New Instance

Figure 8-13 shows the necessary steps to make an operation request on a new **entity** component:

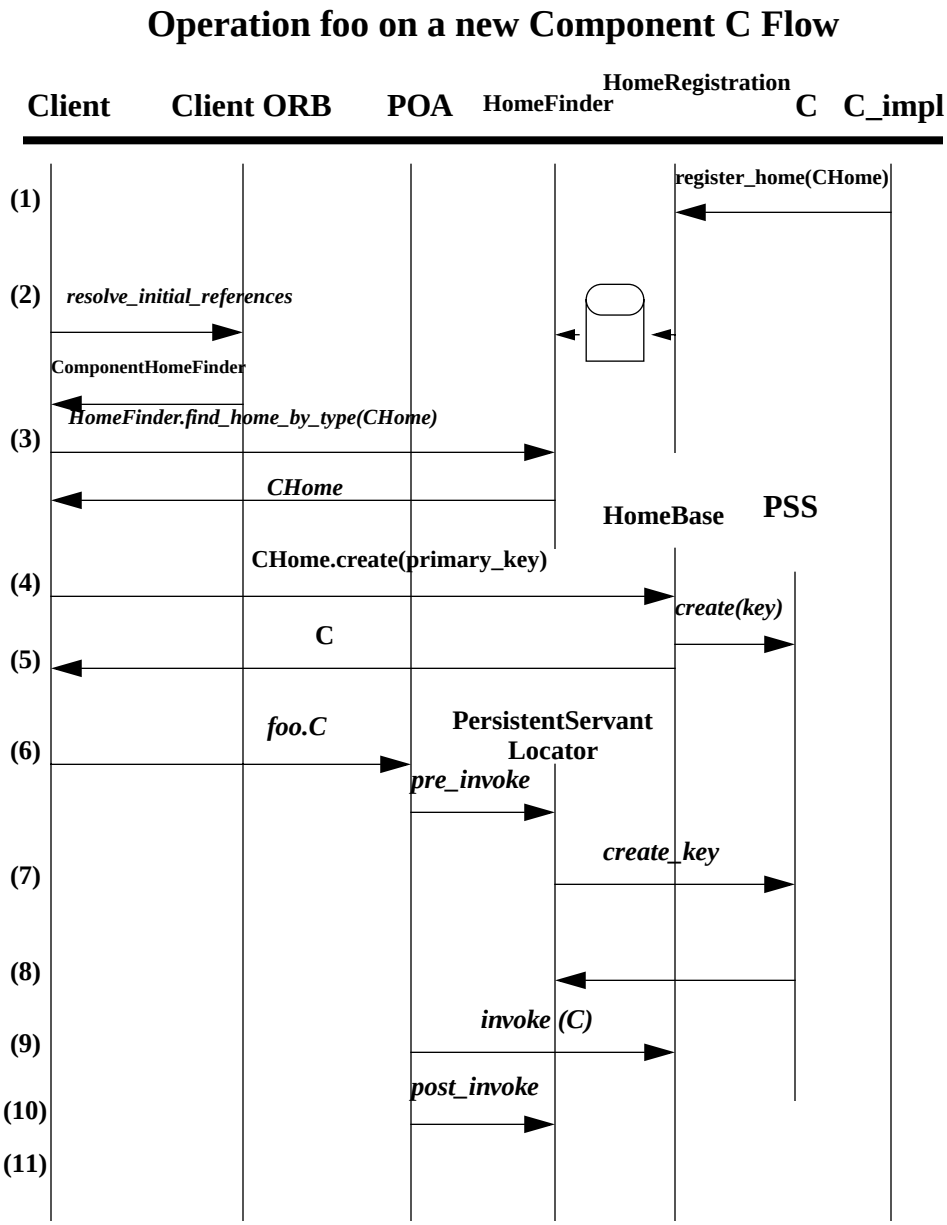

Figure 8-13 Using the Entity Container to Create new Entity Components

1. Component implementation registers the **entity** component home with the **HomeFinder** (**HomeRegistration.register_factory**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.
3. Client uses the **HomeFinder.find_home_by_type** operation to find a factory (**CHome**) that creates component instances of type **C**.
4. Client invokes a **create** operation on the factory (**CHome.create**) using a primary key. Since **C** is an **entity** component, the factory must talk to a persistence mechanism to create a new record in the persistent store using the same primary key.
5. A reference to **C** is returned to the client.
6. Client invokes the **foo** operation on **C** (**C.foo**). Since **C** is not active, the POA invokes the pre_invoke operation on the appropriate **ServantManager** (**PersistentServantLocator.pre_invoke**).
7. The **PersistentServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism finds the appropriate incarnation and returns it to the **PersistentServantLocator**.
8. The **PersistentServantLocator** locates the **ExecutorFactory** (not shown) and creates a new **executor** to handle the request. The associated servant is returned to the POA to process the request.
9. The POA then dispatches the request to the servant (**invoke(C)**)
10. After the request completes, the POA invokes the **PersistentServantLocator (PersistentServantLocator.post_invoke)**.
11. Steps [6] through [10] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **PersistentServantLocator** releases the associated **executor**.

8.3.5.4 Finders and Existing Instances

The **entity** component may also correspond to an existing element in a persistent store. If so, a finder is responsible for locating the **PersistentId** and associating an incarnation with an instance of the **entity** component. The home interface for **entity** components supports finder operations.

The client will use either the **HomeFinder** or the Naming service to locate the home interface. A CORBA component implementation can locate its home interface using the context provided by its container.

8.3.5.5 *Invoking an Operation on an Existing Instance*

Figure 8-14 shows the necessary steps to make an operation request on an existing **entity** component:

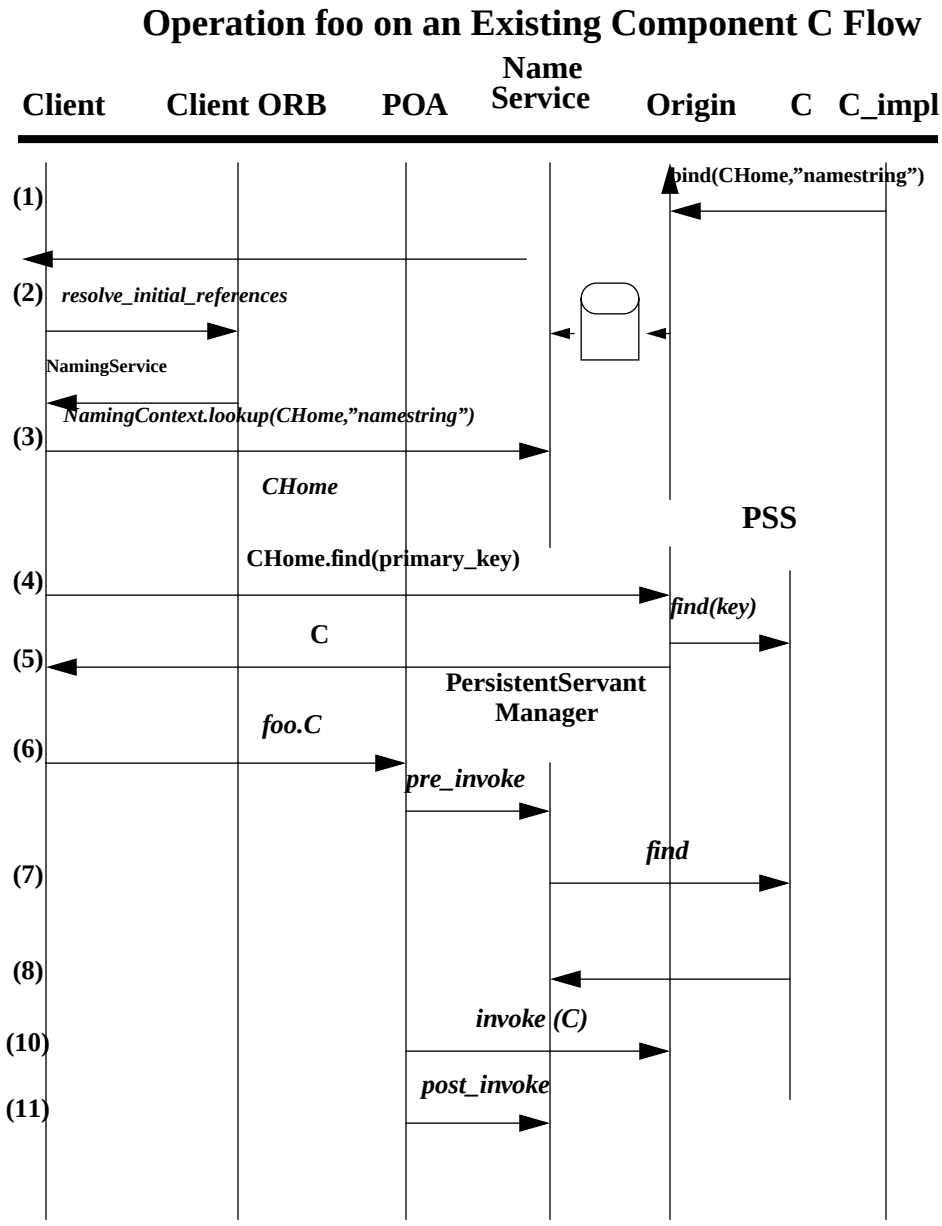

Figure 8-14 Using the Entity Container to Locate Existing Entity Components

1. Component implementation binds the **entity** component home to a string (**"namestring"**) with **CosNaming**.

2. Client uses **ORB.resolve_initial_references** to get a reference to the **NamingService**. Since the **NamingContext** is a righteous CORBA object, its implementation may be located anywhere.
3. Client uses the **NamingContext.lookup** operation to find the home (**CHome**) that finds component instances of type **C**.
4. Client invokes a find operation on the home (**CHome.find**) using a primary key. Since **C** is an **entity** component, the factory must talk to the persistence mechanism to locate an element in the persistent store with the same primary key.
5. A reference to **C** is returned to the client.
6. Client invokes the **foo** operation on **C** (**C.foo**). Since **C** is not active, the POA invokes the pre_invoke operation on the appropriate **ServantManager** (**PersistentServantLocator.pre_invoke**).
7. The **PersistentServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism find the appropriate incarnation and returns it to the **PersistentServantLocator**.
8. The **PersistentServantLocator** locates the **ExecutorFactory** (not shown) and creates a new **executor** to handle the request. The associated servant is returned to the POA.
9. The POA then dispatches the request to the servant (**invoke(C)**)
10. After the request completes, the POA invokes the **PersistentServantLocator (PersistentServantLocator.post_invoke)**.
11. Steps [6] through [10] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **PersistentServantLocator** releases the associated **executor** to the pool.

8.3.5.6 Servant Lifetime Management

The **entity** container supports multiple servant lifetime policies.Support for multiple servant lifetime policies is equivalent to the **process** container as described in Section 8.3.4.4 on page 207.

8.4 Persistence Integration

Component containers support persistence for the **fwork-b container type**. The container architecture permits the persistence provider to be separate from the container provider since we expect that these functions will often be provided by different vendors. This section describes the various forms of persistence support available for CORBA components and the responsibilities of the container, the persistence provider, and the component developer.

The **fwork-b container type** provides two forms of component persistence:

- Container-managed persistence where the container provider interacts with the persistence provider and
- Component-managed persistence where the component developer must interact with the persistence provider.

These are described more fully in the subsequent sections.

8.4.1 Container Managed Persistence

Container-managed persistence supports the declaration of abstract state associated with the component and/or its provided interfaces. This abstract state is declared using the **storage** declaration defined in Chapter 6. State which is to be container-managed can use a CORBA persistence provider or it may use some other persistence mechanism. When CORBA persistence is used, code can be generated to support the **activate** and **passivate** operations on the **fwork-bComponent** interface. If CORBA persistence is not used, the component developer must implement these operations as well as provide implementations for all factory and finder methods defined on the storage's home.

With container-managed persistence, it is still possible for the component to augment the abstract state definition with its own private state. The **load** and **store** operations on **fwork-bComponent** support this function. The container also supports the **Storage** interface which provides run time access to the primary key (for **entity** components) and to the required persistence functions implemented by a persistence provider to enable the component to save and restore its private state. If other mechanisms are used for persistence, it is the responsibility of the component developer to provide the required access.

8.4.2 Component Managed Persistence

Component-managed persistence is also provided by the **fwork-b container type**. Component-managed persistence is selected by not declaring abstract state for a particular component. With component-managed persistence, automatic code generation for saving and restoring state is not possible, so the responsibility lies completely with the component developer. Again, the developer may chose between a CORBA persistence provider and its own defined persistence mechanism. The **Storage** interface provides run time access to the persistence provider. The component developer must use the operations on **fwork-bOrigin** to create a **ComponentId** that encapsulates the location of the persistent state. These operations are defined in Section 7.4.4.3 on page 169.

8.4.3 Interactions between the Container and the Persistence Provider

The design for CORBA components assumes the likelihood that containers and persistence solutions will be provided by different vendors. The impact of that assumption on the generated code was covered in Chapter 6. This assumption also effects both the component developer and the container provider. The component

developer is isolated from the persistence provider by the **Storage** interface defined in Section 7.4.4.4 on page 172. The container provider has several other responsibilities for persistence integration. These include:

- establishing connection to the persistence mechanism,
- managing DB connections with the persistence store, and
- synchronizing component state with durable state.

These subjects are covered in the next sections.

8.4.3.1 Connecting to the Persistence Mechanism

As part of creating the **fwork-b container type**, connectivity to the persistence mechanism must be established. This includes getting any initial references which the persistence provider makes available through the ORB, connecting to the persistence provider (including exchanging any security information required), and obtaining references from the persistence provider to allow the operations on the container's **Storage** interface to be delegated to the proper interface in the persistence service implementation. We assume most of this information will come from container-specific configuration data, although some of it may be standardized if and when the OMG adopts a specification for CORBA persistence.

> *By ensuring that the component developer has access to all required persistence functions, the unspecified configuration data effects only the container implementation. This can be elaborated if and when a CORBA persistence specification is adopted.*

8.4.3.2 Managing DB Connections

Most persistence providers today require that a DB connection be allocated by a client before any data access operations can be invoked. Typically, this is a very expensive process, which must be done infrequently to achieve reasonable system performance. We expect container implementations to manage a pool of such connections, which they have initially created as part of the container creation process and allocate these to the component implementation as needed, typically for the duration of a transaction, although a connection may be retained longer if the container does not need it for some other component. As a result, component implementations will not have to deal with this function directly. Since the container is wrapping the persistence API (using the **Storage** interface), the DB connection can be assigned to a component when its initial request to the persistence provider is made.

8.4.3.3 Synchronization of Component State with Persistence State

The **Storage** interface defines a **flush** operation (on **StorageHome**) which can be used by the component developer to transfer state from the container domain to the persistence domain. For component-managed persistence, the component developer assumes this responsibility as part of the **store** callback operation. For container-managed persistence, the container assumes this responsibility. Two strategies are possible depending on the functions supported by the persistence provider:

- the generated code which implements the **passivate** callback invokes the **flush** operation for each segment of the component.
- the container itself invokes the **flush** operation for a specific transaction if the persistence provider supports that functionality.

Either technique guarantees that the persistence provider, and not the component developer or the container, assumes the responsibility for durability of persistent state.

8.5 Event Management Integration

CORBA components define a simple event model which supports two forms of event communication:

- events which can only be provided by a single supplier
- events which are published anonymously for any subscriber

The container is responsible for mapping those semantics on to the CORBA notification service. Although it is possible to connect event consumers and suppliers directly (to support the first case), delivering such events through a notification channel ensure a more robust event distribution mechanism and allows transaction semantics defined with the event deployment descriptor to be applied to both the delivery of the event to the channel and the removal of the events from the channel.

A component event is represented as a CORBA **valuetype**. This permits event emitters and publishers to be matched with their consumers by the event types they wish to exchange. The event architecture as described in Section 5.7 on page 52 requires that the **valuetype** be able to be transmitted as a CORBA **any** through an event channel. This makes it possible for the container to use untyped channels for transmitting the actual event. The containers responsibility can be broken into three major area and is summarized in the next few sections:

- setting up the channels to be used, including all required proxies
- accepting a CORBA component event and pushing it to an event channel as a structured event
- receiving an event from an event channel and converting it to a CORBA component event

8.5.1 Channel setup

When a component is installed in a container, the deployment descriptor contains information about the types of events it can emit and whether they are intended for general consumption or are targeted explicitly for another component. The container is responsible for initializing the CORBA notification service and establishing the event channels to be used. The actual channel names are not defined in the deployment descriptors and must be made available to the container in a container-specific configuration file. This allows the installation to configure shared channels to be used by other notification service users as well as component implementations. The

container must create a unique channel for events which are designated as emanating from this component only. The technique by which uniqueness is ensured is not specified.

> *There are several possible schemes that could be made to work. Channels could be given unique names using something like a UUID to ensure uniqueness. Hierarchical names is another possibility, where all channels created by a specific container would be prefixed by the name of the container (perhaps a URL). CORBA Security could also be used to prevent events from being pushed to a channel which is dedicated to component events. Other schemes are also possible.*

The CORBA notification service supports filters on both the supply side and the consume side of a channel and allows them to be configured on the channel itself, or on the proxy being used to supply or consume events. This specification allows the container provider to setup filters in any way it chooses since they too must be made available to the container at start-up through a container-specific configuration file.

8.5.2 Transmitting an event

When a CORBA component emits or publishes an event (using the **push** operation on **<event_ type>Consumer**), the operation is delegated to the container by the generated code so that the container can actually push this event to the proper channel. The following steps are required:

- channel lookup - for published events, this is the channel configured for general use at container start-up, for emitted events, this is the channel established by the container for the purpose of pushing this event type.
- Constructing the notification **EventHeader** - The EventHeader consists of some static information, including the **event_type** and **event_name** (not to be confused with the **<event_type>** of the CORBA **valuetype** which holds the event). For CORBA components the **event_type** is set to the value of **<event_type>** and the **event_name** is set to **<source_name>**.
- If configuration-defined filterable data is to be associated with this event, it is placed in the portion of the structured event header defined by the CORBA notification service (**CosNotification::FilterableEventBody**). Container implementations are not required to insert filerable data.
- The **valuetype** representing the actual event data is placed into the **any** portion of the structured event.
- A **CosNotifyCom::push_structured_event** is issued to CORBA notification.

8.5.3 Receiving an event

In order to receive an event, the container must connect its proxy to the event channel the event is to be received on and implement the **CosNotifyComm::structured_push_consumer** interface. The container connects to the channel as a result of a either a **subscribe_<source_name>** or a **connect_<source_name>** operation on the component interface. The container

performs a **CosNotifyChannelAdmin::connect_structured_push_consumer** operation on behalf of the component. The subscribe operation receives all events from the channel, subject to filter constraints; the connect operation receives only those events emitted by the component supplier.

When the container's **structured_push_consumer** interface is invoked, it performs the following processing:

- It extract the event data from the **any** portion of the structured event and converts it to a CORBA **valuetype** which represents the event.
- It extracts the **event_type** and **event_name** from the **FixedEventHeader** and converts them back to the **<event_type>** name of the component event
- It invokes **<event_type>Consumer::push** passing in the **valuetype <event_type>.**

8.6 Servant Locators for CORBA Components

This specification defines two derived **ServantLocator** interfaces to implement the two **container types**:

- The **TransientServantLocator** is used to implement the **fwork-a container type** which supports **service** and **session component categories**.
- The **PersistentServantLocator** is used to implement the **fwork-b container type** which supports **process** and **entity component categories**.

8.6.1 The TransientServantLocator

The **ServantManager** for a **fwork-a container type** is the **TransientServantLocator**. The **TransientServantLocator** implements both the **service** and **session** component interfaces defined in Section 7.4.3 on page 163. The **TransientServantLocator** supports the use of a pool of pre-registered **executors** and the creation of new **executors** and their associated servants on demand using a **ExecutorFactory**. The **ExecutorFactory** has an **ExecutorFactoryId** which corresponds to an interface name. An **executor** and its associated servant (whether created dynamically by the **ExecutorFactory** or statically by the **fwork-a** container) can be shared across a set of **ObjectIds** associated with the same **ServantId** (typically an interface name).

The **TransientServantLocator** is used with both the **poa-x** and **poa-y container implementation types**, which means it will support both variations of POAs. This is depicted in Figure 8-15 below:

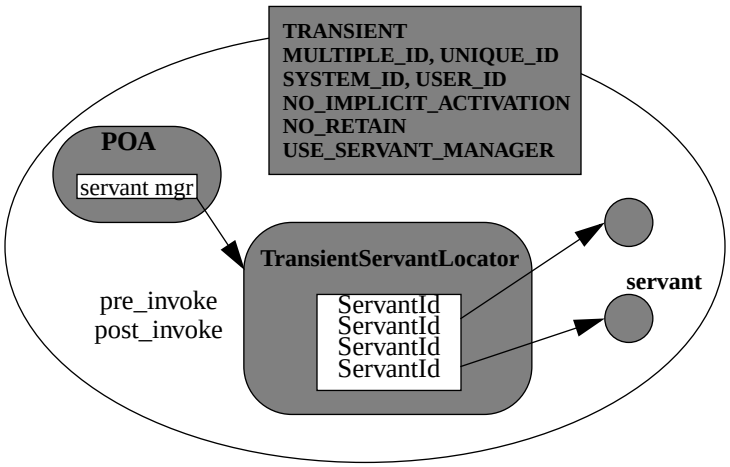

Figure 8-15 TransientServantLocator

The **fwork-a** container is responsible for allocating component instances from a pool of equivalent, reusable instances which are created and managed by the **TransientServantLocator**, the specialized **ServantManager** for the **fwork-a container type**. The **TransientServantLocator** is derived from **PortableServer::ServantLocator** and provides operations for managing **executors** and their associated servants for both **service** and **session** components.

```
native ExecutorFactory;
typedef sequence<octet> ExecutorFactoryId;
typedef sequence<octet> ServantId;

interface TransientServantLocator : PortableServer::ServantLocator {
    PortableServant::Servant register_servant (
        in ServantId sid,
        in PortableServer::Servant svt);
    void unregister_servant (
        in PortableServer::Servant svt);
    PortableServer::Servant lookup_servant (in ServantId sid);
    ExecutorFactory register_executor_factory (
        in ExecutorFactoryId sfid,
        in ExecutorFactory factory);
    void unregister_executor_factory (in ExecutorFactoryId sfid);
    ExecutorFactory lookup_executor_factory (
        in ExecutorFactoryId sfid);
};
```

register_servant

The **register_servant** operation associates a **ServantId** (**sid**) and a servant (**svt**) with a specific POA. When subsequent requests arrive for that servant, the servant identified by **sid** will be dispatched to process them. Although the use of **sid** is not specified, a typical use by a container implementation could be to treat **sid** as an interface name, allowing a group of servants to be used based on the interface identifier.

unregister_servant

The **unregister_servant** operation removes the servant **(svt)** from the pool of registered servants. If the servant identified by **svt** is not registered, it has no effect. The **register_servant** and **unregister_servant** operations allow a pre-allocated collection of servants to be managed as a pool.

lookup_servant

The **lookup_servant** operation is used to allocate a particular servant (identified by **sid**) to process this request. It returns a **PortableServer::Servant** to be used by the POA.

register_executor_factory

The **register_executor _factory** operation associates a **ExecutorFactoryId** (**sfid**) with a specific **ExecutorFactory** (**factory**). The **ExecutorFactory** will be subsequently located by its **ExecutorFactoryId** to create a servant to process subsequent requests. A typical use would be to define a **ExecutorFactory** (**sfid**) for each interface name. The **ExecutorFactory** is implemented by the **InternalHome**.

unregister_executor_factory

The **unregister_executor_factory** operation removes the **ExecutorFactory** (identified by **sfid**) from the list of registered **ExecutorFactory**. If the **ExecutorFactory** identified by **sfid** is not registered, it has no effect.

lookup_executor_factory

The **lookup_executor_factory** operation is used to locate a particular **ExecutorFactory** (by **sfid**). The **ExecutorFactory** is used to create a servant to process this request.

8.6.2 The PersistentServantLocator

The **ServantManager** for the **fwork-b container type** is the **PersistentServantLocator**. The **PersistentServantLocator** implements both the **process** and **entity** component interfaces defined in Section 7.4.4 on page 166. The **PersistentServantLocator** associates an **ObjectId** with a dedicated **executor**. The

servant which supports the **ObjectId** is identified to the container after it creates an object reference and before it is exported for client use. This is depicted in Figure 8-16 below:

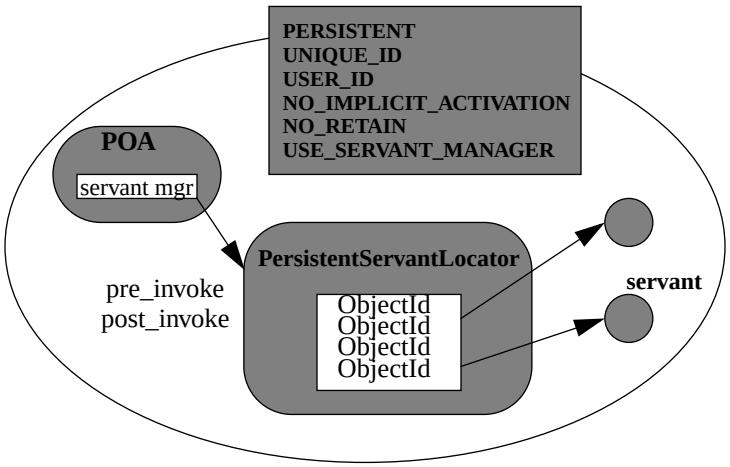

Figure 8-16 PersistentServantLocator

The **fwork-b container type** is responsible for both creating unique object references and creating and managing component instances to support those references. These instances are created on demand and managed by the **PersistentServantLocator**, the specialized **ServantManager** for the **fwork-b container type**. The **PersistentServantLocator** is derived from **PortableServer::ServantLocator** and supports the creation of **ObjectId**s usable with or without the CORBA persistence service. The **PersistentServantLocator** is defined by the following IDL:

```
native ExecutorFactory;
typedef sequence<octet> ExecutorFactoryId;

interface PersistentServantLocator :
    PortableServer::ServantLocator {
        PortableServer::ObjectId create_oid_from_cid (
            in ExecutorFactoryId sfid,
            in Server::ComponentId cid);
        Server::ComponentId get_cid_from_oid (
            in PortableServer::ObjectId);
        ExecutorFactory register_executor_factory (
            in ExecutorFactoryId sfid,
            in ExecutorFactory factory);
        void unregister_executor_factory (in ExecutorFactory Id sfid);
        ExecutorFactory lookup_executor_factory (
            in ExecutorFactoryId sfid);
        void register_servant (in PortableServer::Server svt);
};
```

create_oid_from_cid

The **create_oid_from_pid** operation creates an **ObjectId** that includes state to be managed by some persistence mechanism. The state declaration is defined by the **ComponentId** (**cid**) which encapsulates either a **PersistentId** for CORBA persistence or an **ApplId** for other persistence mechanisms, and the **ExecutorFactory** (**sfid**) is the factory for creating a servant of this type. The **ExecutorFactory** is implemented by the **InternalHome**.

get_cid_from_oid

The **get_cid_from_oid** operation retrieves a **ComponentId** interface (**cid**) from the **ObjectId** (**oid**). The **ComponentId** (**cid**) encapsulates either a **PersistentId** for CORBA persistence or an **ApplId** for other persistence mechanisms and is used to locate the component's persistent state.

register_executor_factory

The **register_executor_factory** operation associates a **ExecutorFactory** (**factory**) with a **ExecutorFactoryId** (**sfid**).The **ExecutorFactory** (**factory**) is responsible for creating a servant of the required type.

unregister_executor_factory

The **unregister_executor_factory** operation removes the **ExecutorFactory** (**sfid**) from the list of registered factories. If the **ExecutorFactory** identified by **sfid** is not registered, it has no effect.

lookup_executor_factory

The **lookup_executor_factory** operation is used to locate a particular **ExecutorFactory** (**sfid**) to create an **executor** and its associated servant of the required type. It returns a **ExecutorFactory** which can create a servant for use by the POA.

register_servant

The **register_servant** operation is used to register a servant **(svt)** created by a **ExecutorFactory** with the POA.

Packaging and Deployment 9

Component implementations may be packaged and deployed.

A CORBA Component package represents one or more implementations of an abstract component. It may be installed on a computer or grouped together with other components to form an *assembly.* A component assembly is a group of interconnected components represented by an assembly package.

A package, in general, consists of a descriptor and a set of files. The descriptor describes the characteristics of the package and points to its various files. The files that make up a package, including the descriptor may be grouped together in an archive file or stored separately. When stored separately, the descriptor contains pointers to the location of each file.

The component package is a special kind of software package. This software packaging scheme, described here, could be used to package arbitrary software entities. In fact it is inspired by the Open Software Description (OSD) proposal to the W3C from Marimba and Microsoft. OSD is an XML vocabulary for describing software packages and their dependencies. We have extended OSD slightly, without loss of generality, to support component packaging.

A component package may be deployed, as is, or it may be included in a component assembly package and deployed as part of the assembly.

A component assembly is a set of interrelated components and component homes represented by an assembly package. A component assembly package consists of a set of component packages and an assembly descriptor. The assembly descriptor specifies the components that make up the assembly, partitioning constraints, and connections. Connections are between *provides* and *uses* ports and *emits* and *consumes* ports. The assembly package is used as input to a deployment tool.

A deployment tool deploys individual components and assemblies of components to an installation site.The user of the tool guides where each component should be installed. The components within an assembly may be installed on a single machine or scattered across a network.

The deployment tool uses installation objects on each host to install and activate component instances. It then configures each component's properties and connections.

9.1 Change History

The major changes to the Packaging and Deployment chapter are:

1. Changed name of "**componentinstance"** element in assembly descriptor to "**componentplacement"**. The use of component instance was confusing, especially since we use the term elsewhere in the document to mean an instance of a component type.
2. Added deployment details.

Changes in March 1st draft:

1. Added properties file descriptor.
2. Changed name of "container" descriptor to "CORBA component" descriptor.
3. Expanded the scope of the component descriptor.
4. Specified Java and C++ entry points.
5. Reorganized descriptor sections to list elements alphabetically.
6. Added some elements to the chapter which were defined in the DTD but not in this text.

9.2 Component Packaging

A CORBA Component implementation is described by a software package. A software package is represented by a descriptor and a set of files. The descriptor and associated files may be loosely coupled or grouped together in a ZIP archive file. These software archive files are distinguished by a ".car" extension.

9.3 Software Package Descriptor

CORBA component implementations are described by a software package descriptor. The descriptor consists of general information about the software followed by one or more sections describing implementations of that software. An XML vocabulary, derived from the Open Software Description proposal, is used to describe component software packages. The descriptor file has a ".csd" extension. CSD stands for CORBA Software Descriptor. When used in an archive, the CSD file for the archive is placed in a top level directory called "meta-inf".

The structure and intent of the descriptor can be better understood by looking at an example.

9.3.1 A softpkg Descriptor Example

```
<softpkg name="Bank" version="1,0,1,0">
    <pkgtype>CORBA Component</pkgtype>
    <title>Bank</title>
    <author>
        <company>Acme Component Corp.</company>
        <webpage href="http://www.acmecomponent.com/>
    </author>
    <description>Yet another bank example</description>
     <license href="http://www.acmecomponent.com/license.html" />
     <idl id="IDL:M1/Bank:1.0" ><link href="ftp://x/y/Bank.idl"/></idl>

     <propertyfile><fileinarchive name="bankprops.cpf"/></propertyfile>

     <implementation id="DCE:700dc518-0110-11ce-ac8f-0800090b5d3e">
        <os name="WinNT" version="4,0,0,0" />
        <os name="Win95" />
        <processor name="x86" />
        <compiler name="MyFavoriteCompiler" />
        <programminglanguage name="C++" />

       <dependency type="ORB"><name>ExORB</name></dependency>

        <descriptor type="CORBA Container">
            <fileinarchive>processcontainer.ccd</fileinarchive>
       </descriptor>

       <code type="DLL">
            <fileinarchive name="bank.dll"/>
            <entrypoint>createBankHome</entrypoint>
       </code>

        <threadsafety level="class"/>

       <dependency type="DLL">
            <localfile name="rwthr.dll"/>
       </dependency>

    </implementation>

    <implementation> <!-- another implementation --> </implementation>
</softpkg>
```

9.3.2 The Software Package Descriptor XML Elements

This section describes the XML elements that make up a software package descriptor. The section is organized starting with the root element of the package descriptor document, **softpkg**, followed by all subordinate elements, in alphabetical order. The complete softpkg DTD may be found in the appendix.

9.3.2.1 The softpkg Root Element

The **softpkg** element is the root element of the document. As well, it is a child element of **dependency**. It contains a set of general child elements that describe the software package. This is followed by one or more implementation specifications.

A **softpkg** archive may contain multiple implementations of a component. This might be necessary to specialize implementations for different operating systems, compilers, or ORBs, or to provide different programming language implementations of the component. Each implementation is represented in the **softpkg** descriptor as a distinct implementation element.

```
<!ELEMENT softpkg
        ( title
        | pkgtype
        | author
        | description
        | license
        | idl
        | propertyfile
        | dependency
        | descriptor
        | implementation
        | extension
        )* >
<!ATTLIST softpkg
        name        ID      #REQUIRED
        version     CDATA #OPTIONAL >
```

The attributes are as follows:

name

Uniquely identifies the package within the package.

version

Specifies the version of the component. The format of the version string is numerical major and minor version numbers separated by commas (e.g., “1,0,0,0”).

9.3.2.2 The author Element

The **author** element is used to identify the author of the **softpkg**. It may contain **name**, **company**, and **webpage** child elements.

```
<!ELEMENT author
        ( name
        | company
        | webpage
        )* >
```

9.3.2.3 The code Element

The **code** element points to a file in the archive which implements the component. This could be, for example, a DLL, a .so, or a .class file. The **fileinarchive** child element is used to indicate the code file within the archive. **codebase** and **link** are used to point to code files outside of any archive. The optional **entrypoint** child element is used to specify an entry point or usage of the code.

```
<!ELEMENT code
        ( codebase
        | fileinarchive
        | link
        )
        , entrypoint?) >
<!ATTLIST code
        type CDATA #IMPLIED
```

The **type** specifies the type of code.

9.3.2.4 The codebase Element

The **codebase** element is used to specify a resource. If the resource isn't available in the local environment, then a link specifies where it may be obtained. **codebase** has an EMPTY content model.

```
<!ELEMENT codebase EMPTY >
<!ATTLIST  codebase
        filename CDATA #IMPLIED
        %simple-link-attributes; >
```

codebase has two attributes: **name** - the name of the resource, and **href**--as defined in **simple-link-attributes**--the link.

9.3.2.5 The company Element

The **company** element, an optional child element of **author**, specifies the company that created the **softpkg**. It contains string data.

```
<!ELEMENT company ( #PCDATA ) >
```

9.3.2.6 The compiler Element

The optional **compiler** element specifies the compiler used to create an implementation. **compiler** has an empty content model.

```
<!ELEMENT compiler EMPTY >
<!ATTLIST compiler
        name    CDATA #REQUIRED
        version CDATA #IMPLIED >
```

The required attribute **name**, specifies the name of the compiler and the optional **version** the version of the compiler. The version is specified in a “w,x,y,z” format.

9.3.2.7 The dependency Element

The **dependency** element is used to specify environmental or other dependencies. The type of dependency is specified by the **type** attribute. The **dependency** element is a child element of both the **softpkg** element and **implementation** elements. When used as a child of **softpkg**, it specifies general dependencies applicable to all implementations. When used as a child of **implementation**, it specifies implementation specific dependencies.

```
<!ELEMENT dependency
        ( softpkg
        | codebase
        | fileinarchive
        | localfile
        | name
        ) >
<!ATTLIST dependency
        type   CDATA              #IMPLIED
        action (assert | install)"assert">
```

The **type** attribute specifies the type of the resource required. This may be set to, for example, “DLL”, “.so”, or “.class”.

When **action** is set to **assert**, the installation process must verify that the dependency exists in the environment. If **action** is set to **install**, the installation process must install the dependency if it does not already exist.

9.3.2.8 The description Element

The **description** element contains a string description. It is used to describe its parent element. It contains string content.

```
<!ELEMENT description ( #PCDATA ) >
```

9.3.2.9 The descriptor Element

The **descriptor** element is used to refer to descriptor files associated with a **softpkg** or **implementation**. In a CORBA Component **softpkg**, it is used to point to the container descriptor.

```
<!ELEMENT descriptor
        ( link
        | fileinarchive
        ) >
<!ATTLIST descriptor
        type   CDATA  #IMPLIED>
```

9.3.2.10 The entrypoint Element

The **entrypoint** element specifies the entry point to a software package. See section 9.9.8 for information on CORBA component entry points.

```
<!ELEMENT entrypoint ( #PCDATA ) >
```

9.3.2.11 The extension Element

The **extension** element is used to add experimental or vendor specific elements to the **softpkg** DTD. The content model of the extension element is **ANY**, meaning that it can have character data or any combination of element types defined in the DTD, in any order.

An effort has been made to make the **extension** element an optional child element of all non-trivial elements. Processors may ignore **extension** elements that they do not recognize.

```
<!ELEMENT extension ANY >
<!ATTLIST extension
        class       CDATA     #REQUIRED
        origin      CDATA     #REQUIRED
        id          ID        #IMPLIED
        extra       CDATA     #IMPLIED
        html-form   CDATA     #IMPLIED >
```

The attributes of the **extension** element are as follows:

class

Used to distinguish this extension element usage. A processing application identifies extension elements that it understands by examining an extension element's **class** and **origin** attributes.

origin

An **origin** attribute is required to identify the party responsible for the extension; for example, an ORB vendor.

id

An optional ID attribute which must be unique in the file.

extra

An *extra* attribute that may be used however the originator wishes.

html-form

The **html-form** element is used for formatting. The content will be formatted per the html element type indicated, e.g., "".

9.3.2.12 The fileinarchive Element

The **fileinarchive** element is used to specify a file in the same archive as the descriptor. The optional **link** element may be used to point to an external archive, in which case the file will be looked for in that file.

```
<!ELEMENT fileinarchive
        ( link? ) >
<!ATTLIST fileinarchive
        name CDATA #REQUIRED >
```

The **name** attribute specifies the name or path of the element in the archive.

9.3.2.13 The idl Element

The **idl** element points to the description of the interface that the **softpkg** implements.

The **idl** element specifies the **id** of the component and the file or repository where it is defined.

```
<!ELEMENT idl (link | fileinarchive | repository) >
<!ATTLIST idl
   id CDATA #REQUIRED>
```

The **id** attribute specifies the **id** for the component; e.g., a repository id.

9.3.2.14 The implementation Element

The **implementation** element contains descriptive information about a particular implementation of the software represented by the **softpkg** descriptor. An implementation is described by platform dependencies, container policies, dependencies, code filename, entry points and other characteristics.

```
<!ELEMENT implementation
        ( description
        | code
        | compiler
        | dependency
        | descriptor
        | extension
        | programminglanguage
        | humanlanguage
        | os
        | propertyfile
        | processor
        | runtime
        | threadsafety
        )* >
<!ATTLIST implementation
         id ID #IMPLIED >
```

The **id** attribute is a DCE UUID which uniquely identifies the implementation.

9.3.2.15 The license Element

The **license** child element of **softpkg** is used to point to the text of a usage license. The license is pointed to by an **href** attribute. The **license** element may have arbitrary string content.

```
<!ELEMENT license ( #PCDATA ) >
<!ATTLIST license
        %simple-link-attributes; >
```

9.3.2.16 The link Element

The **link** element is used to specify a generic link. The **href** attribute indicates the link. The element can have string content.

```
<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
        %simple-link-attributes; >
```

9.3.2.17 The localfile Element

The **localfile** element is used to specify a file that is expected to be found in the local environment.

```
<!ELEMENT localfile EMPTY >
<!ATTLIST localfile
        name CDATA #REQUIRED >
```

The name of the file is specified in the **name** attribute.

9.3.2.18 The name Element

The **name** element, as an optional child element of **author**, specifies the name of the author. It has string content.

```
<!ELEMENT name ( #PCDATA ) >
```

9.3.2.19 The naturallanguage Element

The **naturallanguage** element specifies a human language. **naturallanguage** has an EMPTY content model.

```
<!ELEMENT naturallanguage EMPTY >
<!ATTLIST naturallanguage
        name CDATA #REQUIRED >
```

The natural language name is specified in the **name** attribute.

9.3.2.20 The os Element

The **os** element is used to specify a particular operating system that the implementation will work with. This can be specified multiple times if the implementation will work on more than one **os**.

```
<!ELEMENT os EMPTY >
<!ATTLIST os
        name    CDATA #REQUIRED
        version CDATA #IMPLIED>
```

The **name** attribute specifies the name of the operating system.

The **version** attribute specifies the version of the **os** in “w,x,y,z” format.

Legal values include:

- AIX
- BSDi
- VMS
- DigitalUnix
- DOS
- HPBLS
- HPUX
- IRIX
- Linix
- MacOS
- OS/2
- AS/400
- MVS
- SCO CMW
- SCO ODT
- Solaris
- SunOS
- UnixWare
- VxWorks
- Win95
- WinNT

9.3.2.21 The pkgtype Element

The **pkgtype** element is used to identify the type of software that the **softpkg** represents. This specification reserves package types "**CORBA Component**" and "**CORBA Interface Impl**" for the packaging of CORBA component and interface implementations.

```
<!ELEMENT pkgtype ( #PCDATA ) >
```

9.3.2.22 The processor Element

The **processor** element indicates the type of processor that the implementation must run on, if there is any such constraint.

```
<!ELEMENT processor EMPTY >
<!ATTLIST processor
        name CDATA #REQUIRED >
```

The name of the processor is indicated in the **name** attribute.

Legal values include:

- x86
- mips
- alpha
- ppc
- sparc
- 680x0
- vax
- AS/400
- S/370

9.3.2.23 The programminglanguageElement

The **programminglanguage** element specifies the type of the component implementation. **programminglanguage** has an empty content model. **programminglanguage** is a child element of **implementation**.

```
<!ELEMENT programminglanguage EMPTY>
<!ATTLIST programminglanguage
        name CDATA #REQUIRED
        version  CDATA #IMPLIED >
```

The required programminglanguage **name** and optional **version** attributes specify the programming language used to implement the component.

9.3.2.24 The propertyfile Element

The **propertyfile** element is used to refer to a property file associated with the **softpkg** or implementation.

```
<!ELEMENT propertyfile
    ( fileinarchive
    | link) >
```

9.3.2.25 The repository Element

The **repository** element is used to refer to an interface repository in which the idl definition can be found. **repository** has an *empty* content model.

```
<!ELEMENT repository EMPTY >
<!ATTLIST repository
        type CDATA #IMPLIED
         %simple-link-attributes; >
```

The interface repository location is specified by an **href** attribute (as specified in the **simple-link-attributes** entity). An optional type attribute is allowed if the type of **repository** is different than the default for the given component.

9.3.2.26 The runtime Element

The **runtime** element specifies a runtime required by a component implementation. An example of a runtime is a Java VM.

```
<!ELEMENT runtime EMPTY >
<!ATTLIST runtime
        name    CDATA #REQUIRED
        version CDATA #IMPLIED>
```

The name and version of the runtime are specified in the **name** and **version** attributes. The version is specified in “w,x,y,z” format.

9.3.2.27 The softpkg Element

This is the root element of the descriptor. See section 9.3.2.1.

9.3.2.28 The simple-link-attributes Entity

The **simple-link-attributes** entity is used to specify link attributes. The default link form is a simple link.

```
<!ENTITY % simple-link-attributes "
        xml:link        CDATA                   #FIXED 'SIMPLE'
        href            CDATA                   #REQUIRED
        ">
```

The user of an element that uses these link attributes will likely only need to be concerned with the **href** attribute. However the user may specify other attributes if desired.

To demonstrate the usage of an element that employs the **simple-link-attributes** entity, consider the following element definition:

```
<!ELEMENTexampleelement EMPTY >
<!ATTLIST   exampleelement
        %simple-link-attributes; >
```

This could be used as follows:

```
<exampleelement href="http://www.abc.com/xyz" />
```

Issue – The W3C XLL work is still in progress at the time of this writing. This entity definition will be modified if necessary when the W3C work completes.

9.3.2.29 The threadsafety Element

The **threadsafety** element is used to indicate the level of thread safety of the implementation. **threadsafety** has an empty content model.

```
<!ELEMENT threadsafety EMPTY >
<!ATTLIST threadsafety
        level (none|class|instance) #REQUIRED >
```

The thread safety is specified in the level attribute to be either none, class, or instance.

none - means that no thread safety can be assumed.

class - means that the code is reentrant and that any global, static, or class data will be protected when accessed by multiple threads.

instance - means that instance data of the component will be protected when simultaneously accessed by multiple threads.

9.3.2.30 The title Element

The **title** element is used to specify the friendly, or tool name of the **softpkg**. The title element contains string data.

```
<!ELEMENT title ( #PCDATA ) >
```

9.3.2.31 The webpage Element

The **webpage** element, an optional child element of **author**, specifies a web page associated with the author.

```
<!ELEMENT webpage ( #PCDATA ) >
<!ATTLIST webpage
        %simple-link-attributes; >
```

9.4 CORBA Component Descriptor

The CORBA Component descriptor describes a component. It is referred to by a **<descriptor type="CORBAContainer>** element in a softpkg descriptor describing a CORBA component.The CORBA Component descriptor specifies component characteristics, used at assembly and deployment time.

The component descriptor provides information at component assembly time about the interfaces that a component supports and its ports. For the purpose of component packaging and deployment we will define ports as the interfaces that the component *uses* and *provides* and the events that it *emits* and *consumes*. A component descriptor file has a recommended ".ccd" extension, standing for CORBA Component Descriptor.

At deployment time, the component descriptor is used to determine the type of container in which the component needs to be installed.

The component descriptor has two main parts. The first part describes general information about the component and characteristics of the component that are important at deployment time.

The second part of the component descriptor describes the structure of the component with respect to supported interfaces, inherited components, and uses and provides ports. This information allows a tool to display the features of a component and to connect components together based on those features. For example, a component which *uses* interface Z could be *connected* to another component that *provides* Z based on information in the two components descriptors.

The component descriptor is generated by the CIDL compiler. This is convenient as the CIDL compiler has much of the necessary information at hand. However, the compiler doesn't have all of the information required. The user will have to modify the generated descriptor. This could be done manually, but it is more likely to be done with the help of a tool.

The component descriptor is described using an XML vocabulary. The complete XML DTD for the descriptor is in the appendix. This chapter will discuss each element in detail. But before we do that, familiarize yourself with the following example.

9.4.1 CORBA Component Descriptor Example

```
<corbacomponent>
  <corbaversion> 3.0 </corbaversion>
  <repositoryid id="IDL:BookStore:1.0" />
  <componentkind>
    <entity>
      <servant lifetime="process" />
      <persistence responsibility="container" usepss="true">
        <persistentstoreinfo
          implementation="acmepss"
          datastorename="oracle"
          datastoreid="mainofficedb" />
      </persistence>
    </entity>
  </componentkind>
  <transaction use="supports" />
  <eventpolicy emit="normal" />
  <threading policy="multithread" />
  <configurationcomplete set="true" />

  <componentfeatures name="BookStore" repid="IDL:BookStore:1.0">
    <inheritscomponent repid="IDL:Acme/Store:1.0" />
    <ports>
      <provides
        providesname="book_search"
        repid="IDL:BookSearch:1.0" />
      <provides
        providesname="shopping_cart"
        repid="IDL:CartFactory:1.0" />
      <uses
        usesname="ups_rates"
        repid="IDL:ShippingRates:1.0" />
      <uses
        usesname="fedex_rates"
        repid="IDL:ShippingRates:1.0" />
      <emits
        emitsname="low_stock"
        eventtype="StockRecord"
        eventname="LowStock" />
    </ports>
  </componentfeatures>

  <componentfeatures name="Store" repid="IDL:Acme/Store">
    <supportsinterface repid="IDL:Acme/GeneralStore" />
    <ports>
      <provides
        providesname="admin"
        repid="IDL:Acme/StoreAdmin:1.0" />
    </ports>
  </componentfeatures>
```

```
    <interface name="BookSearch" repid="IDL:BookSearch:1.0">
      <inheritsinterface repid="IDL:SearchEngine:1.0" />
    </interface>
    <interface name="SearchEngine" repid="IDL:SearchEngine:1.0"/>
    <interface name="CartFactory" repid="IDL:CartFactory:1.0"/>
    <interface name="ShippingRates" repid="IDL:ShippingRates:1.0"/>
    <interface name="StoreAdmin" repid="IDL:Acme/StoreAdmin:1.0"/>
    <interface name="GeneralStore" repid="IDL:Acme/GeneralStore:1.0"/>
</corbacomponent>
```

9.4.2 The CORBA Component Descriptor XML Elements

This section describes the XML elements that make up a component descriptor. The section is organized starting with the root element of the component descriptor document, **corbacomponent**, followed by all subordinate elements, in alphabetical order. The complete CORBA component descriptor DTD may be found in the appendix.

9.4.2.1 The corbacomponent Root Element

The **corbacomponent** element is the root element of the CORBA component descriptor.

```
<!ELEMENT corbacomponent
    ( corbaversion
    , repositoryid
    , componentkind
    , transaction
    , security?
    , eventpolicy?
    , threading
    , configurationcomplete
    , extendedpoapolicy*
    , repository?
    , componentfeatures+
    , interface*
    , extension*
    ) >
```

These elements must be provided in the order presented.

- **corbaversion** tells which version of CORBA the component is assuming.
- **repositoryid** is the interface repository id of the component. It also refers to a **componentfeatures** element later in the descriptor.
- **componentkind** describes properties of the component which will determine what kind of container the component must reside in.
- **security** determines security policies.

- **eventpolicy** determines policies for emitted and consumed events.
- **threadingpolicy** determines whether calls to the component will be serialized or not.
- **configurationcomplete** is set if the component expects for configuration_complete to be called on the component.
- **extendedpoapolicy** is used to set a poa policy for the component beyond the base poa policies. For example firewall policies.
- **repository** provides a reference to a repository, such as the interface repository.
- **componentfeatures** describes component inheritance, supported interfaces, uses and provides ports, and emits and consumes ports of the component. If the primary component inherits from other components, those components are described in separate **componentfeature** elements.
- **interface** describes the simple name and repository id of an interface and points to inherited interfaces. Between the **componentfeatures** and **interface** elements, one can navigate all of the interfaces that a component uses, provides, supports, and inherits.
- **extension** may be used by a user or vendor to provide proprietary information in the component descriptor.

These are the top-level elements of the document. These descriptor elements are described in terms of attributes and other elements. The remainder of this section will describe the top-level and child elements in detail.

Elements are presented in alphabetical order so that they will be easy to locate.

Note – A good strategy for examining an XML DTD is to recursively navigate from the root element, which in this case is **corbacomponent**, to each child element.

See appendix B.2 for the full text of the component descriptor DTD. Read ahead for information on individual elements.

9.4.2.2 The client Element

Child element of **securitycredentialkind**.

The credentials obtained by the client making the invocation are associated with the operation. This restricts authorization to operations permitted to the client by the security administrator. This is the default security policy.

<!ELEMENT client EMPTY >

9.4.2.3 The componentfeatures Element

Child element of **corbacomponent**.

The **componentfeatures** element is used to describe a component with respect to the components that it inherits from, the interfaces that the component supports, and its provides, uses, emits, and consumes ports. A component has the features that it supports directly, plus the features that it inherits through other components. Additionally, supported interfaces may inherit from other interfaces. In essence a graph is formed from the primary component to a set of ports, supported interfaces, and other components. The primary component is identified by the repositoryid child element of corbacomponent.

The information obtained by traversing the componentfeatures graph may be displayed by graphical tools. But more importantly, it allows component assembly tools to make decisions about what interfaces on a component (supported or provided) are available to connect to uses ports on other components.

```
<!ELEMENT componentfeatures
   ( inheritscomponent ?
   , supportsinterface*
   , ports
   , extension*
   ) >
<!ATTLIST componentfeatures
   name CDATA #REQUIRED
   repid ID #REQUIRED >
```

The **name** attribute is the non-qualified name of the component.

The **repid** attribute is the fully qualified repository id of the component. repid is also used to refer to this component from elsewhere in the descriptor, for example from the inheritscomponent element).

9.4.2.4 The componentkind Element

Child element of **corbacomponent**.

The componentkind element defines the component category. For more information on these categories, see the Container chapter.

```
<!ELEMENT componentkind
    ( service
    | session
    | process
    | entity
    | unclassified
    ) >
```

9.4.2.5 The configurationcomplete Element

Child element of **corbacomponent**.

The **configurationcomplete** attribute is used to set whether configuration_complete should be called on the component after it has been fully configured.

```
<!ELEMENT configurationcomplete EMPTY >
<!ATTLIST configurationcomplete
    set ( true | false ) #REQUIRED >
```

9.4.2.6 The consumes Element

Child element of **ports**.

A consumes port specifies an event that the component expects to receive. At deployment or creation time, the component will be connected via a channel to other components or entities that emit the event.

```
<!ELEMENT consumes EMPTY>
<!ATTLIST consumes
    consumesname CDATA #REQUIRED
    eventtype CDATA #REQUIRED
    eventname CDATA #REQUIRED >
```

consumesname

The **consumesname** attribute identifies the name associated with the consumes statement in idl.

eventtype

The **eventtype** attribute identifies the **eventtype** as defined in CORBA notification that the component wishes to subscribe to.

eventname

The **eventname** attribute identifies the **eventname** as defined in CORBA notification that the component wishes to subscribe to.

9.4.2.7 The corbacomponent Element

The root element of this CORBA Component descriptor. See section 9.4.2.1.

9.4.2.8 The corbaversion Element

Child element of **corbacomponent**.

The corbaversion is used to identify the version of CORBA that the component implementation is assuming. The version is represented by a major and minor number separated by a “.”. For example, “<corbaversion>3.0</corbaversion>”.

```
<!ELEMENT corbaversion (#PCDATA) >
```

9.4.2.9 The emits Element

Child element of **ports**.

A emits port specifies an event that the component generates. At deployment or creation time, the component will be connected to a channel in which it can be connected to consuming components.

```
<!ELEMENT emits EMPTY>
<!ATTLIST emits
   emitsname CDATA #REQUIRED
   eventtype CDATA #REQUIRED
   eventname CDATA #REQUIRED >
```

emitsname

The **emitsname** attribute identifies the name associated with the emits statement in idl.

eventtype

The **event_type** attribute identifies the **event_type** as defined in CORBA notification that the component wishes to subscribe to.

eventname

The **event_name** attribute identifies the **event_name** as defined in CORBA notification that the component wishes to subscribe to.

9.4.2.10 The entity Element

Child element of **componentkind**.

The **entity** component kind is described in Section 7.2.4 on 140.

```
<!ELEMENT entity
   ( servant
   , persistence
   ) >
```

9.4.2.11 The eventpolicy Element

Child element of **corbacomponent**.

Event policies define the quality of service associated with events emitted or consumed by this component. Event policies can also be used to statically define the specific events the component intends to consume.

```
<!ELEMENT eventpolicy EMPTY>
<!ATTLIST eventpolicy
    emit ( normal | default | transaction ) #IMPLIED
    consume (normal | default | transaction ) #IMPLIED >
```

The possible values are defined Section 7.3.8 on 146

9.4.2.12 The extendedpoapolicy Element

Child element of **corbacomponent**.

The **extendedpoapolicy** element is a name-value pair used to specify POA policies beyond the base set of policies. It is for new policies, such as firewall, or future POA policies yet to be defined. The extendedpoapolicy element must not be used to specify any of the base poa policies. A set of poa policies is predefined for each component category, except for the unclassified category. Only the unclassified component type is flexible with respect to base poa policies; these are set using the **poapolicies** child element of the unclassified element.

```
<!ELEMENT extendedpoapolicy EMPTY>
<!ATTLIST extendedpoapolicy
    name CDATA #REQUIRED
    value CDATA #REQUIRED >
```

The **name** attribute is the name of the poa policy as defined in the specification where it originated.

The **value** attribute is a valid attribute for the policy as defined in the specification where it originated.

9.4.2.13 The extension Element

Child element of **corbacomponent, componentfeatures**.

The extension element is used to add vendor or user specific information to the component descriptor.

```
<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
    class    CDATA    #REQUIRED
    origin   CDATA    #REQUIRED
    id       ID       #IMPLIED
    extra    CDATA    #IMPLIED
    html-form CDATA    #IMPLIED >
```

The **class** attribute defines the category of extension, what would be an element type if it could be added to the descriptor.

origin identifies the organization or group responsible for the extension.

id, **extra**, and **html-form** are optional attributes that the originator of the extension may wish to employ.

9.4.2.14 The inheritscomponent Element

Child element of **componentfeatures**.

The inheritscomponent element specifies an inherited component.

```
<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
   repid IDREF #REQUIRED>
```

The **repid** identifies is the repository id of the inherited component, and it is used to refer to the componentfeatures element of the inherited component, elsewhere in the descriptor.

9.4.2.15 The inheritsinterface Element

Child element of **interface**.

The inheritsinterface element is used to specify interface inheritance. This allows, for example, for a derivation chain to be followed from a supported or provided interface up to but excluding the Object interface.

```
<!ELEMENT inheritsinterface EMPTY>
<!ATTLIST inheritsinterface
   repid IDREF #REQUIRED>
```

The **repid** identifies is the repository id of the inherited interface, and it is used to refer to the interface element of the inherited interface, elsewhere in the descriptor.

9.4.2.16 The ins Element

Child element of **repository**.

The ins element is used to specify an interoperable naming service name.

```
<!ELEMENT ins EMPTY>
<!ATTLIST ins
    name CDATA #REQUIRED >
```

name is the INS name.

9.4.2.17 The interface Element

Child element of **corbacomponent**.

Specifies an interface that the component, either directly or through inheritance, provides, uses, or supports.

```
<!ELEMENT interface ( inheritsinterface* ) >
<!ATTLIST interface
   name CDATA #REQUIRED
   repid ID #REQUIRED >
```

The **name** attribute is the non-qualified name of the interface.

The **repid** attribute is the fully qualified repository id of the interface. repid is also used to refer to this interface from elsewhere in the descriptor, for example from the inheritsinterface element).

9.4.2.18 The objref Element

Child element of **repository**.

The **objref** element is used to specify a stringified object reference.

```
<!ELEMENT objref EMPTY>
<!ATTLIST objref

    string CDATA #REQUIRED >
```

The **string** attribute holds the stringified object reference.

9.4.2.19 The persistence Element

Child element of **entity, process**.

The **persistence** element is used to define persistence parameters: whether the component assumes container of component managed persistence and whether the CORBA Persistent State Service (PSS) is to be used.

```
<!ELEMENT persistence ( persistentstoreinfo? )>
<!ATTLIST persistence
    responsibility ( container | component ) #REQUIRED
    usepss ( true | false ) #REQUIRED >
```

The **responsibility** attribute is used to specify container of component managed persistence.

The **usepss** attribute determines whether the PSS will be used or not.

9.4.2.20 The persistentstoreinfo Element

Child element of **persistence, unclassified**.

The **persistentstoreinfo** element is used to specify a persistence implementation, , the type of datastore, and a particular datastore.

```
<!ELEMENT persistentstoreinfo EMPTY>
<!ATTLIST persistentstoreinfo
    implementation CDATA #REQUIRED
    datastorename CDATA #REQUIRED
    datastoreid CDATA #REQUIRED >
```

The **implementation** attribute identifies a particular persistence implementation such as a particular vendor's PSS implementation.

The **datastorename** attribute identifies the type of datastore, for example a particular vendor's database.

The **datastoreid** identifies a particular instance of a datastore, for example the name of a database file.

9.4.2.21 The poapolicies Element

Child element of **unclassified**.

The poapolicies element is used to identify poa creation parameters for an empty container in which an *unclassified* category component will reside.

```
<!ELEMENT poapolicies EMPTY>
<!ATTLIST poapolicies
    thread (ORB_CTRL_MODEL | SINGLE_THREAD_SAFE ) #REQUIRED
    lifespan (TRANSIENT | PERSISTENT ) #REQUIRED
    iduniqueness (UNIQUE_ID | MULTIPLE_ID) #REQUIRED
    idassignment (USER_ID | SYSTEM_ID) #REQUIRED
    servantretention (RETAIN | NON_RETAIN) #REQUIRED
    requestprocessing (USE_ACTIVE_OBJECT_MAP_ONLY
                |USE_DEFAULT_SERVANT
                |USE_SERVANT_MANAGER) #REQUIRED
    implicitactivation (IMPLICIT_ACTIVATION
                |NON_IMPLICIT_ACTIVATION) #REQUIRED >
```

The poapolicies attributes are as defined in the base POA specification.

Note – Not all combinations of POA policies are valid. A good tool component packaging tool will not permit the user to specify invalid POA policy combinations. In case an invalid combination of policies is set on a the empty container, the container/POA should throw an exception.

9.4.2.22 The ports Element

Child element of **componentfeatures**.

The **ports** element describes what interfaces a component provides and uses, and what events it emits and consumes. Any number of uses, provides, emits, and consumes elements can be specified in any order.

```
<!ELEMENT ports
    ( uses
    | provides
    | emits
    | consumes
    )* >
```

9.4.2.23 The process Element

Child element of **componentkind**.

The **process** component kind is described in Section 7.2.4 on 140.

```
<!ELEMENT process
    ( servant
    , persistence
    ) >
```

9.4.2.24 The provides Element

Child element of **ports**.

The **provides** element specifies an interface that is provided by the component.

```
<!ELEMENT provides EMPTY>
<!ATTLIST provides
    providesname CDATA #REQUIRED
    repid IDREF #REQUIRED >
```

The **providesname** is the name given to the provides port in IDL.

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

9.4.2.25 The repository Element

Child element of **corbacomponent**.

The repository element is used to point to a repository, such as the interface repository.

```
<!ELEMENT repository ( ins | objref ) >
<!ATTLIST repository
   type CDATA #IMPLIED >
```

The **type** attribute specifies the type of repository. Currently, the only predefined value for **type** is “CORBA Interface Repository”.

9.4.2.26 The repositoryid Element

Child element of **corbacomponent**.

repositoryid identifies the repository id of the component described by this descriptor. The repository id also serves to point to the primary **componentfeatures** element for this component within the descriptor, so as to distinguish it from inherited components.

```
<!ELEMENT repositoryid EMPTY >
<!ATTLIST repositoryid
   repid IDREF #IMPLIED >
```

repid is the fully qualified repository id of the component.

9.4.2.27 The security Element

Child element of **corbacomponent**.

Security policies define the relationship between the client identity and the credentials associated with an operation invocation as seen by the CORBA security service. These credentials allow security to be applied to the CORBA component without application awareness.

```
<!ELEMENT security
   ( securitycredentialkind ) >
```

The possible values are defined Section 7.3.7 on 146.

9.4.2.28 The securitycredentialkind Element

Child element of **security**.

The **securitycredentialkind** is used to specify how credentials are associated with operations on the component. Three possible policies are defined.

```
<!ELEMENT securitycredentialkind
    ( client
    | system
    | specified
    ) >
```

9.4.2.29 The servant Element

Child element of **entity, process, session**.

Servant lifetime policies control the lifetime of the servant which implements a component's operations and provide an aid to efficiently manage storage of components within a server process. Servant lifetime policies are fixed for **service** components. Servant lifetime policies must be specified for **session**, **process** and **entity** components and are implemented by the component using APIs provided by the container.

```
<!ELEMENT servant EMPTY >
<!ATTLIST servant
    lifetime (process|method|transaction) #REQUIRED >
```

The possible values are defined in Section 7.3.5 on 143.

9.4.2.30 The service Element

Child element of **componentkind**.

Specifies that the component is of the **service** category. The service component kind is described in chapter 7.

Issue – Fix container chapter reference.

```
<!ELEMENT service EMPTY >
```

9.4.2.31 The session Element

Child element of **componentkind**.

Specifies that the component is of the **session** category. The session component category is described in chapter 7.

Issue – Fix container chapter reference.

```
<!ELEMENT session
   ( servant ) >
```

9.4.2.32 The specified Element

Child element of **securitycredentialkind**.

The credentials associated with **userid** are associated with operations invocations. This restricts authorizations to operations permitted to the user specified by **userid**.

```
<!ELEMENT specified EMPTY >
<!ATTLIST specified
   userid CDATA #REQUIRED >
```

9.4.2.33 The supportsinterface Element

Child element of **componentfeatures**.

Identifies an interface that the component supports, as defined in IDL.

```
<!ELEMENT supportsinterface EMPTY>
<!ATTLIST supportsinterface
   repid IDREF #REQUIRED >
```

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

9.4.2.34 The system Element

Child element of **securitycredentialkind**.

The credentials associated with the container that houses the CORBA component are associated with operation invocations. In general this will allow a higher degree of privilege than that granted to the client. This is particularly useful when a connection to a database needs to be made on behalf of all clients using the container.

```
<!ELEMENT system EMPTY >
```

9.4.2.35 The threading Element

Child element of **corbacomponent**.

The threading element determines the threading policy of the container in which it is placed.

```
<!ELEMENT threading EMPTY>
<!ATTLIST threading
    policy ( serialize | multithread ) #REQUIRED >
```

Setting the threading **policy** to **serialize** means that the container will serialize calls to the container; in this case, the implementation must have a threadsafety level of class or instance.

Setting the threading **policy** to **multithread** means that multiple threads of control can be active in the component at one time; in which case, the implementation must have a threadsafety level of instance.

9.4.2.36 The transaction Element

Child element of **corbacomponent**.

The Transaction Policy attribute controls the way transactions are managed by the container for this component. Six possible values can be selected by the component developer to provide maximum flexibility.

```
<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
    use (not-supported|required|supports|requires-new|mandatory|never)
#REQUIRED >
```

The possible values for **use** are defined Section 7.3.6 on 144.

9.4.2.37 The unclassified Element

Child element of **componentkind**.

The **unclassified** element identifies that the component is of the unclassified sort. See the container chapter for more information on the unclassified component category.

Issue – Fix container chapter reference.

```
<!ELEMENT unclassified
    ( poapolicies
    , persistentstoreinfo
    ) >
```

9.4.2.38 The uses Element

Child element of **ports**.

The **uses** element specifies an interface that is used by the component, as specified in an IDL uses specification.

```
<!ELEMENT uses EMPTY>
<!ATTLIST uses
    usesname CDATA #REQUIRED
    repid IDREF #REQUIRED >
```

The **usesname** is the name given to the uses port in IDL.

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

9.5 Component Assembly Packaging

If a component package is the vehicle for deploying a single component implementation, then a component assembly package is the vehicle for deploying a set of interrelated component implementations. It is a template or pattern for instantiating a set of components and introducing them to each other.

An assembly package consists of a descriptor and a set of component packages and property files. These files may by packaged together in an archive file or distributed. When distributed, the descriptor represents the package and holds links to its associated files.

The component assembly descriptor describes which components make up the assembly, how those components are partitioned, and how they are connected to each other. A component assembly descriptor is the recipe for deploying a set of interconnected components.

An assembly is normally created visually within a design tool, however it is possible to create assemblies using more primitive tools.

Note – An assembly specifies an *initial* configuration. The actual connected graph of components may evolve beyond that initial configuration. The assembly does not address the evolution of this graph.

9.6 Component Assembly File

The component assembly archive file is a ZIP file containing a component assembly descriptor, a set of component archive files, and, if necessary, a set of component property files. The component assembly archive file has a ".aar" extension.

9.7 Component Assembly Descriptor

A component assembly descriptor is specified using an XML vocabulary. Each component assembly package must contain a single descriptor file. Component descriptors have a ".cad" extension. CAD stands for Component Assembly Descriptor.

The assembly descriptor describes a component assembly. It consists of elements describing component implementations used in the assembly, connection information, and partitioning information.

Component "placements" are particular uses of a component implementation. A component placement may have a specialized property file.

Components, as reference by component placements, are connected by their *provides* and *uses* interfaces, or by their *emits* and *consumes* events. If one component provides an interface of a particular type and another component uses an interface of that type, then we can pass the reference of the provided interface to the component that uses it, in effect connecting the two components. In the same way, we connect two components where one emits an event that the other consumes.

Sets of component instances may be partitioned. Components may be free or partitioned to a generic set of hosts and processes. This is really a process of conveying that specific components are to be collocated within a single process or host. Free components, components that are not used in a collocation may be deployed in any manner at deployment time.

When used in an archive, the CAD file for the archive is placed in a top level directory called “meta-inf”.

9.7.1 Component Assembly Descriptor Example

The following example illustrates how to write a component assembly descriptor. For further information, see the element descriptions that follow and the XML DTDs in the appendix.

Issue – Nail down the DOCTYPE specification.

```
<!--
<!DOCTYPE componentassembly PUBLIC "-//OMG//DTD
componentassembly v1.0 //EN"
"http://www.omg.org/dtds/componentassembly1_0.dtd">
-->
<!DOCTYPE componentassembly SYSTEM "componentassembly.dtd">
<componentassembly id="ZZZ123">
    <componentfiles>
        <componentfile id="A">
            <fileinarchive name="ca.car"/>
        </componentfile>
        <componentfile id="B">
            <fileinarchive name="cb.car"/>
        </componentfile>
        <componentfile id="C">
            <fileinarchive name="cc.car">
                <link href="ftp://www.xyz.com/car/cc.car"/>
            </fileinarchive>
        </componentfile>
        <componentfile id="D">
            <fileinarchive name="cd.car"/>
        </componentfile>
        <componentfile id="E">
            <fileinarchive name="ce.car"/>
        </componentfile>
        <componentfile id="F">
            <fileinarchive name="cf.car"/>
        </componentfile>
    </componentfiles>

    <partitioning>

        <componentplacement id="Aa">
            <componentfileref idref="A"/>
        </componentplacement>

        <processcollocation cardinality="*">
            <usagename>Example process collocation</usagename>
            <programminglanguage name="C++" />  <!-- optional -->
            <componentplacement id="Bb">
                <componentfileref idref="B"/>
            </componentplacement>
            <componentplacement id="Cc">
                <componentfileref idref="C"/>
            </componentplacement>
        </processcollocation>

        <hostcollocation cardinality="1">
            <usagename>Example host collocation</usagename>
            <processcollocation cardinality="*">
                <componentplacement id="Dd">
```

```
                    <componentfileref idref="D"/>
                </componentplacement>
                <componentplacement id="Ee">
                    <componentfileref idref="E"/>
                </componentplacement>
            </processcollocation>
            <componentplacement id="Ff" cardinality="*">
                <componentfileref idref="F"/>
            </componentplacement>
        </hostcollocation>

        <componentplacement id="Aaa">
            <usagename>Example placement</usagename>
            <componentfileref idref="A"/>
            <componentimplref idref="ghi"/> <!-- optional -->
            <propertiesfile>
                <fileinarchive name="aProperties.cdr"/>
            </propertiesfile>
            <registerwithnaming name="sink"/>
            <registerwithtrader>
                <traderproperties>
                    <traderproperty>
                        <traderpropertyname>rate</traderpropertyname>
                          <traderpropertyvalue>10</traderpropertyvalue>
                    </traderproperty>
                </traderproperties>
            </registerwithtrader>
        </componentplacement>

    </partitioning>

    <connections>
        <connectinterface>
            <usingcomponent idref="Aa">
                <usesidentifier>abc</usesidentifier>
            </usingcomponent>
            <providingcomponent idref="Bb">
                <providesidentifier>abc</providesidentifier>
                <findby><namingservice/></findby>
            </providingcomponent>
        </connectinterface>
        <connectevent>
            <emitingcomponent idref="Ee">
                <emitsidentifier>mno</emitsidentifier>
                <findby><stringifiedobjectref/></findby>
            </emitingcomponent>
            <consumingcomponent idref="Aaa">
                <consumesidentifier>pqr</consumesidentifier>
            </consumingcomponent>
        </connectevent>
    </connections>
```

```
</componentassembly>
```

9.7.2 Component Assembly Descriptor XML Elements

This section describes the XML elements that make up a component assembly descriptor. The section is organized starting with the root element of the descriptor document, **componentassembly**, followed by all subordinate elements, in alphabetical order. The complete component assembly DTD may be found in the appendix.

9.7.2.1 The componentassembly Root Element

The **componentassembly** element is the root element of the component assembly descriptor. It has three child elements that serve to define the assembly: **componentfiles**, **partitioning** and **connections**. The **extension** element can be used to add proprietary or experimental elements to the component assembly document.

```
<!ELEMENT componentassembly
        ( componentfiles
        | partitioning
        | connections
        | extension
        )* >
<!ATTLIST componentassembly
        id ID #IMPLIED >
```

The **id** attribute specifies an identifier for the **componentassembly**.

Issue – Should this be a UUID?

9.7.2.2 The codebase Element

See section 9.3.2.4.

9.7.2.3 The componentfile Element

The **componentfile** element refers to a component archive file or a software descriptor. The component file may be part of the component assembly archive or external to the archive. **componentfile** elements are referenced by **componentplacement** elements.

componentfile contains either a **fileinarchive, link** or **codebase** element.

```
<!ELEMENT componentfile
        ( fileinarchive
        | codebase
        | link
        ) >
<!ATTLIST componentfile
        id ID #REQUIRED >
```

The **id** attribute must uniquely identify the **componentfile** element within the descriptor.

9.7.2.4 The componentfileref Element

The **componentfileref** element refers to a particular **componentfile** element in the **componentfiles** block.

```
<!ELEMENT componentfileref EMPTY >
<!ATTLIST componentfileref
        idref IDREF #REQUIRED >
```

The **idref** attribute corresponds to a unique **componentfile id** attribute.

9.7.2.5 The componentfiles Element

The **componentfiles** element is used to list all of the component files that are used in the assembly. At least one component file must be listed.

Each component file is uniquely identified and referred to by component instances used in the assembly. Multiple component instances may refer to a single component file.

```
<!ELEMENT componentfiles
        ( componentfile+
        ) >
```

9.7.2.6 The componentimplref Element

The **componentimplref** element is used to refer to a particular implementation in a component file.

```
<!ELEMENT componentimplref EMPTY >
<!ATTLIST componentimplref
        idref CDATA #REQUIRED >
```

The **idref** attribute refers to a unique implementation **id** in the component descriptor.

9.7.2.7 The componentplacement Element

This **componentplacement** element describes a particular deployment of a component instance or home. The **componentplacement** element may be a direct child of the **partitioning** element which states that it has no collocation constraints; or a **componentplacement** may be a child element of the **hostcollocation** or **processcollocation** elements which states a collocation constraint with other component placements.

The **usagename** child element is used to specify a name for the placement, possibly for use in a tool. The **stringifiedobjectref** element is used in the construction of a specific object reference. The **componentfileref** element specifies the component file. The **componentimplref** element refers to a specific implementation in the component file. The **propertiesfile** element refers to a state file associated with this placement. The **registerwithnaming** element instructs the installation process to register this component or home with a naming service. The **registerwithtrader** element instructs the installation process to register this component or home with a trader service.

```
<!ELEMENT componentplacement
        ( usagename?
        , componentfileref
        , componentimplref?
        , propertiesfile?
        , stringifiedobjectref?
        , registerwithnaming*
        , registerwithtrader*
        , extension*
        ) >
<!ATTLIST componentplacement
        id          ID      #IMPLIED
        cardinality CDATA "1" >
```

The **id** attribute is a unique identifier within the assembly descriptor for the component. The **id** is used to refer to the component instance in the connect block.The **cardinality** attribute specifies how many instantiations of this component may be deployed. Possible values for cardinality are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1 and there are any connections to this **componentplacement** then connections will be made to each instance of the component or component home.

The **id** attribute is a unique identifier within the assembly descriptor.

9.7.2.8 The connectevent Element

The **connectevent** element is used as a child of the **connections** element to specify a connection between two components based on emitted and consumed events.

The **emitingcomponent** element refers to the component emiting the event. The **consumingcomponent** refers to the component that receives the event.

```
<!ELEMENT connectevent
        ( emitingcomponent
        , consumingcomponent ) >
<!ATTLIST connectevent
        id ID #IMPLIED >
```

The **id** attribute is a unique identifier within the assembly descriptor.

9.7.2.9 The connectinterface Element

The **connectinterface** element is used as a child element of the **connections** element to specify a connection between two components based on provided and used interfaces.

The **providingcomponent** element refers to the component providing the interface. The **usingcomponent** refers to the component that has a need to use the provided interface.

```
<!ELEMENT connectinterface
        ( usingcomponent
        , providingcomponent ) >
<!ATTLIST connectinterface
        id ID #IMPLIED >
```

The **id** attribute is a unique identifier within the assembly descriptor.

9.7.2.10 The connections Element

The connections element is used to specify connections between component instances. A connection may be made from a provided interface port to a used interface port, or from an emitted event port to a consumed event port, or from a uses interface port to a home interface. It is the matching of a supplier to a consumer.

If the component on one end of a connection has a cardinality greater than 1 or if it is part of a process or host collocation with a cardinality greater than 1 then multiple connections will be realized from or to each instance of the component or home.

The **connections** element is used to specify interface and event connections between component instances in the assembly. The **connectinterface** child element is used to make connections between *uses* and *provides* interfaces. The **connectevent** element is used to make connections between *emits* and *consumes* events.

```
<!ELEMENT connections
        ( connectinterface
        | connectevent
        | extension
        )* >
```

9.7.2.11 The consumesidentifier Element

A child element of **consumingcomponent**, **consumesidentifier** identifies which consumes “port” on the component is to participate in the relationship. The type of the consumes event must match the type of the connected emits event.

```
<!ELEMENT consumesidentifier ( #PCDATA ) >
```

9.7.2.12 The consumingcomponent Element

Specifies the event-consuming side of an event connection relationship.The **consumesidentifier** child element identifies the particular consumes port.

```
<!ELEMENT consumingcomponent
        ( consumesidentifier
        , findby* )>
<!ATTLIST consumingcomponent
        idref IDREF #REQUIRED >
```

The **idref** attribute identifies the consuming component instance.

9.7.2.13 The emitingcomponent Element

Specifies the event-emitting side of an event connection relationship.The **emitsidentifier** child element identifies the particular emits identifier in the component IDL.

```
<!ELEMENT emitingcomponent
    ( emitsidentifier
    , findby* )>
<!ATTLIST emitingcomponent
    idref IDREF #REQUIRED >
```

The **idref** attribute identifies the emitting component instance.

9.7.2.14 The emitsidentifier Element

A child element of **emitingcomponent**, **emitsidentifier** identifies which emits “port” on the component is to participate in the relationship. The type of the emits event must match the type of the connected consumes event.

<!ELEMENT emitsidentifier (#PCDATA) >

9.7.2.15 The extension Element

See section 9.3.2.11.

9.7.2.16 The fileinarchive Element

See section 9.3.2.11.

9.7.2.17 The findby Element

The findby element is used to resolve a connection between two components. It tells the installation agent how to locate the components involved in the relationship. In the simplest case, the installer will know where the components are because it was the one responsible for installing those components. But if the components already exist in the installation environment the installer must know how to locate the said components. It could locate a component in a naming service, in a trader, or by a stringified object reference. The purpose of the findby element is to provide such information.

<!ELEMENT findby
(namingservice
| stringifiedobjectref
| installprocess
| traderquery
| extension
) >

9.7.2.18 The hostcollocation Element

A **hostcollocation** specifies a group of component instances that are to be deployed together to a single host. The child elements are an optional **usagename**, an optional **impltype**, and a list of **processcollocation** and **componentplacement** elements. If **impltype** is specified then each of the component instances must have implementations supporting the implementation type. If **impltype** is not specified, then at deployment time each of the collocated components must have implementations supporting the target deployment platform.

```
<!ELEMENT hostcollocation
        ( usagename?
        , impltype?
        ,    ( componentplacement
             | processcollocation
             | extension
             )+
        ) >
<!ATTLIST hostcollocation
        id          ID      #IMPLIED
        cardinality CDATA "1" >
```

The **id** attribute uniquely identifies this host collocation in the component assembly file. The **cardinality** attribute specifies how many instances of this host collocation may be deployed. Possible values for **cardinality** are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1, and there are connections to components within the **hostcollocation**, then connections will be made to the corresponding components or component homes within each instance of the collocation.

9.7.2.19 The impltype Element

Issue – May not be necessary.

9.7.2.20 The installprocess Element

The installprocess is used to indicate that a component should be locatable by the installing agent.

```
<!ELEMENT installprocess EMPTY >
```

Issue – This could also be implicit by the absence of a findby statement. So this may go away.

9.7.2.21 The link Element

See section 9.3.2.16.

9.7.2.22 The namingservice Element

The naming service element is used to indicate that a component should be found using a naming service.

```
<!ELEMENT installprocess EMPTY >
```

9.7.2.23 The partitioning Element

Component partitioning specifies a deployment pattern of components to generic processes and hosts. The pattern is expressed via collocation constraints. A particular usage of a component instance or a component home is called a component placement. A component placement can be collocated with other component placements in a process. Processes and component placements can be collocated within a logical host. A component placement that is not part of a process or host collocation may be deployed without constraint.

Within a **partitioning** element, component instances are *declared* and component collocation constraints are specified. The **componentplacement** child element specifies a freely deployable component instance or home. The **processcollocation** and **hostcollocation** child elements are used to group component placements together into deployable units.

A component placement may be declared as part of a host or process collocation or by itself. The actual host and process will be determined at deployment time. Component instances, process collocations, and host collocations all have an associated cardinality. The default cardinality is “1”. A cardinality greater than 1 allows or mandates that the deployable unit be deployed multiple times.

```
<!ELEMENT partitioning
        ( componentplacement
        | processcollocation
        | hostcollocation
        | extension
        )* >
```

9.7.2.24 The processcollocation Element

The **processcollocation** element specifies a group of component instances that are to be deployed together to a single process. The child elements are an optional **usagename**, an optional **impltype**, and a list of **componentplacement** elements. If **impltype** is specified then each of the component instances must have implementations supporting the implementation type. If **impltype** is not specified, then at deployment time each of the collocated components have implementations supporting the target deployment platform.

```
<!ELEMENT processcollocation
        ( usagename?
        , impltype?
        ,    ( componentplacement
             | extension
             )+
        ) >
<!ATTLIST processcollocation
        id              ID      #IMPLIED
        cardinality CDATA "1" >
```

The **id** attribute uniquely identifies this process collocation in the component assembly file. The **cardinality** attribute specifies how many instances of this process collocation may be deployed. Possible values for **cardinality** are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1, and there are connections to components within the **processcollocation**, then connections will be made to corresponding components or component homes within each instance of the collocation.

9.7.2.25 The propertiesfile Element

The **propertiesfile** element specifies a property file for a component. If the component file has a default property file in the component package, the component property file overrides the default. The property file may be specified by either a **fileinarchive** or a **codebase** child element. The format of the property file is described in section 9.8.

```
<!ELEMENT propertiesfile
        ( fileinarchive
        | codebase
         ) >
```

9.7.2.26 The providesidentifier Element

A child element of **providingcomponent**, **providesidentifier** identifies which provides “port” on the component is to participate in the relationship. The type of the provided interface must match the type of the connected uses interface.

```
<!ELEMENT providesidentifier ( #PCDATA ) >
```

9.7.2.27 The providingcomponent Element

Specifies the interface-providing side of an interface connection relationship. The **providingidentifier** child element identifies the particular *provides* identifier in the component IDL.

```
<!ELEMENT providingcomponent
        ( providesidentifier
        , findby* )>
<!ATTLIST providingcomponent
        idref IDREF #REQUIRED >
```

The **idref** attribute identifies the providing component instance.

9.7.2.28 The registerwithnaming Element

The registerwithnaming element tells the installer to register a component instance or home with a naming service after it is created.

```
<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming
        name CDATA #REQUIRED >
```

The **name** attribute is the naming service name.

9.7.2.29 The registerwithtrader Element

The registerwithtrader element tells the installer to register a component instance or home with a trader after it is created.

```
<!ELEMENT registerwithtrader
        ( traderproperties ) >
<!ATTLIST registerwithtrader
        tradername CDATA #IMPLIED >
```

9.7.2.30 The stringifiedobjectref Element

The stringifiedobjectref element is used to locate a component by its object reference.

```
<!ELEMENT stringifiedobjectref ( #PCDATA ) >
```

9.7.2.31 Trader elements

The trader elements are used to register a component or home with a trader and to find a component or home using a trader query. The trader elements closely parallel trader functionality in name and purpose.

```
<!ELEMENT traderconstraint ( #PCDATA ) >

<!ELEMENT traderexport
    ( traderservicetypename
    , traderproperties
    ) >

<!ELEMENT traderpolicy
    ( traderpolicyname
    , traderpolicyvalue
    ) >

<!ELEMENT traderpolicyname ( #PCDATA ) >

<!ELEMENT traderpolicyvalue ANY >

<!ELEMENT traderpreference ( #PCDATA ) >

<!ELEMENT traderproperties
    ( traderproperty+ ) >

<!ELEMENT traderproperty
    ( traderpropertyname
    , traderpropertyvalue
    ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ANY >

<!ELEMENT traderquery
    ( traderservicetypename
    , traderconstraint
    , traderpreference?
    , traderpolicy*
    , traderspecifiedprop*
    ) >

<!ELEMENT traderservicetypename ( #PCDATA ) >

<!ELEMENT traderspecifiedprop ( #PCDATA ) >
```

Note – These still need to be explained in text. In the mean time, look at the trader spec. The correspondence should be obvious.

9.7.2.32 The usagename Element

A user defined “friendly” name.

```
<!ELEMENT usagename ( #PCDATA ) >
```

9.7.2.33 The usesidentifier Element

A child element of **usingcomponent**, **usesidentifier** identifies which using “port” on the component is to participate in the relationship. The type of the using interface must match the type of the connected provides interface.

```
<!ELEMENT usesidentifier ( #PCDATA ) >
```

9.7.2.34 The usingcomponent Element

Specifies the interface-using side of an interface connection relationship. The **usingidentifier** child element identifies the particular *uses* port.

```
<!ELEMENT usingcomponent
        ( usesidentifier
        , findby* )>
<!ATTLIST usingcomponent
        idref IDREF #REQUIRED >
```

The **idref** attribute identifies the using component instance.

9.8 Property File Descriptor

The property file details component or home attribute settings. Properties are described using an XML vocabulary described below. The property file is used at deployment time to configure a home or component instance. A configurator uses the property file to determine how to set component and component home property attributes.

The property file may be edited using a text editor or with the help of a GUI tool. A component may be shipped with a set of default properties that may be altered by the end user.

The suggested file extension for property files is “.cpf”, for Component Property File.

9.8.1 Property File Example

The following property descriptor example has 3 properties: **bufferSize**, **niceGuys**, and **sanityTestTime**.. The **bufferSize** parameter is a long type; the **niceGuys** property is a sequence of strings; and the **sanityTestTime** property is a structure of type **timestruct**, containing 3 shorts.

```
<properties>
   <simple name=bufSize type="long">
      <description>Size of Whizitron input buffer</description>
      <value>4096</value>
      <defaultvalue>256</defaultvalue>
   </simple>
   <sequence name="niceGuys" type="sequence<string>">
      <simple type="string"><value>Dave</value></simple>
      <simple type="string"><value>Ed</value></simple>
      <simple type="string"><value>Garrett</value></simple>
      <simple type="string"><value>Jeff</value></simple>
      <simple type="string"><value>Jim</value></simple>
      <simple type="string"><value>Martin</value></simple>
      <simple type="string"><value>Patrick</value></simple>
   </sequence>

   <struct name="sanityTestTime" type="timestruct">
      <description>Time to start daily sanity check</description>
      <simple name="hour" type="short"><value> 24 </value></simple>
      <simple name="minute" type="short"><value> 0 </value></simple>
      <simple name="second" type="short"><value> 0 </value></simple>
   </struct>
</properties>
```

The properties document has 3 major elements: simple, sequence and struct.

The **simple** element describes a single primitive idl type. The **sequence** element corresponds to an IDL sequence, and the **struct** element corresponds to an IDL struct.

9.8.2 Property File XML Elements

This section describes the XML elements that make up a properties file. The section is organized starting with the root element of the properties document, **properties**, followed by all subordinate elements, in alphabetical order. The complete properties file DTD may be found in the appendix.

9.8.2.1 The properties Root Element

The properties element the root element of the properties document. The **properties** element contains an optional description and any combination of **simple**, **sequence**, and **struct** elements.

```
<!ELEMENT properties
    ( description?
    , ( simple
      | sequence
      | struct
      )*
    ) >
```

9.8.2.2 The choice Element

```
<!ELEMENT choice ( #PCDATA ) >
```

The **choice** element is used to specify a valid simple property value.

9.8.2.3 The choices Element

```
<!ELEMENT choices ( choice+ ) >
```

The **choices** element is a list of one or more choice elements.

9.8.2.4 The defaultvalue Element

```
<!ELEMENT defaultvalue ( #PCDATA ) >
```

The **defaultvalue** element is used to specify a default simple property value.

9.8.2.5 The description Element

```
<!ELEMENT description ( #PCDATA ) >
```

The **description** element is used to provide a description of its enclosing element.

9.8.2.6 The properties Element

The root element of the properties file. See section 9.8.2.1.

9.8.2.7 The simple Element

The **simple** element is used to specify an attribute value of a primitive type. **simple** contains a mandatory **value** element, and optional **description**, **choices**, and **defaultvalue** elements.

The **value** element is used to specify the value of the simple type. If the **value** element is empty, the value is deemed unspecified. If the value is unspecified, and there is a **defaultvalue** defined, then the default value will be used.

The **description**, **choices** and **defaultvalue** child elements may be used to provide guidance to the end user in deciding how to set the attributes.

```
<!ELEMENT simple
    ( description?
    , value
    , choices?
    , defaultvalue?
    ) >
<!ATTLIST simple
    name CDATA #IMPLIED
    type ( boolean
        | char
        | double
        | float
        | short
        | long
        | objref
        | octet
        | short
        | string
        | ulong
        | ushort
        ) #REQUIRED >
```

name

The **name** attribute specifies the name of the attribute as it appears in IDL. The name attribute is required, except when the property is used in a sequence.

type

The **type** attribute specifies the type of the corresponding attribute. Property types are either an IDL primitive data type, or an objref.

Note – The objref is in its stringified form in the property element. The stringified object reference is converted into a proper object reference before being assigned to its corresponding attribute.

9.8.2.8 The sequence Element

The **sequence** element is used to represent a sequence of similar types. It may be a sequence of simple types, a sequence of structs, or a sequence of sequences. The order of the sequence elements in the property file is preserved in the constructed sequence. An optional description may be used to describe the sequence property.

```
<!ELEMENT sequence
    ( description?
    , ( simple*
      | struct*
      | sequence*
      )
    ) >
<!ATTLIST sequence
    name CDATA #IMPLIED
    type CDATA #REQUIRED >
```

name

The **name** attribute specifies the name of the sequence as it appears in IDL. The name attribute is required, except when the sequence property is used in another sequence.

type

The **type** attribute specifies the type of the corresponding IDL sequence. The type of each element in the sequence must match the sequence type.

9.8.2.9 The struct Element

The **struct** element corresponds to an IDL structure. It may be composed of simple properties, sequences, or other structs.

```
<!ELEMENT struct
    ( description?
    , ( simple
      | sequence
      | struct
      )*
    ) >
<!ATTLIST struct
    name CDATA #IMPLIED
    type CDATA #REQUIRED >
```

name

The **name** attribute specifies the name of the struct attribute as it appears in IDL. The name attribute is required, except when the structure property is used in a sequence.

type

The **type** attribute specifies the type of the corresponding IDL struct.

9.8.2.10 The value Element

The **value** element is used to specify a simple value.

```
<!ELEMENT value ( #PCDATA ) >
```

9.9 Component Deployment

Components, component homes, and component assemblies are deployed on target hosts in a network using a deployment tool provided by an ORB or tool vendor. The deployment application is a CORBA client which communicates with cooperating CORBA objects on target installation platforms.

The aim of deployment is to install and “hook-up” a logical component topology to a physical computing environment. The deployment is specified by an assembly file, or in the degenerate case, an individual component file. When a component is deployed, either the home for that component is created alone or the home is created and then an instance of the component is created.

Issue – Someone might want to create a component but not register its home with a factory finder.

The basic steps in the deployment process are:

1. Identify on which hosts the components are to be installed. This information will most likely come from an interaction between tool and user. Components are deployed either singly or together with other components as part of a process or host collocation.
2. Install component implementations on each platform where corresponding component instances are to be deployed. If a component implementation, uniquely identified by a UUID, is already installed on a host then it does not have to be installed again.
3. Instantiate components and component homes on particular hosts. The mapping for doing so was determined in step 1.
4. Connect components as specified in the assembly descriptor’s connect block.

Keep in mind that a stand-alone component files may deployed as well as assembly files. In that case, step 4 does not apply. Unless otherwise noted, all interfaces defined in the subsequent sections are in the **Deployment** module which is imbedded within the **Components** module (see Appendix A.1 on 399 for a description of the naming structure proposed by this specification).

9.9.1 Participants in Deployment

The deployment of a component or component assembly is carried out by a deployment application in conjunction with a set of helper objects. The helper objects include component repositories, assembly and component factories, an object representing an assembly itself, and a container.

To familiarize yourself with the deployment participants, and their role in deploying components and assemblies, look at the following class diagram and attendant scenario.

9.9.1.1 Deployment Class Diagram

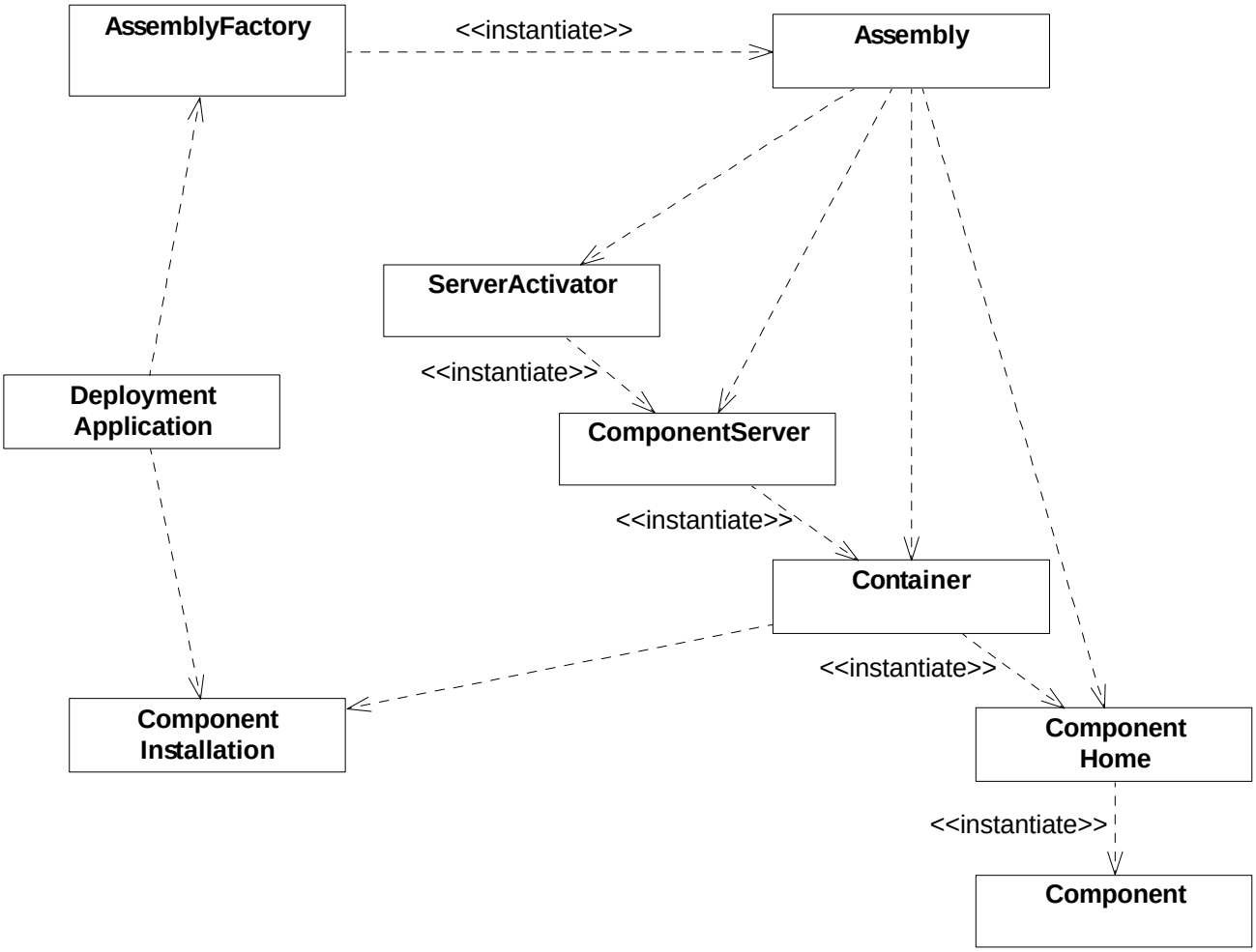

Figure 9-1 Deployment Class (Interface) Diagram

9.9.1.2 Deployment Scenario

Let's go through the steps of deploying and activating a component assembly.

1. The deployment application has a conversation with the user to determine where each component or collocation is to be placed. Information about where components are to be located is recorded in a copy of the component assembly descriptor. This marked-up assembly descriptor will be used later by the Assembly object to direct the creation of the assembly.

2. Next the component implementations are installed on the platforms where they are to be used. The deployment application calls *install* on the **Installation** object, passing the component implementation **id** and a string denoting the address of the

component file. If the component has not already been installed on the target platform, then the **Installation** object retrieves the component file and makes it available in the local environment.

3. The deployment application then creates an **Assembly** object. Assembly objects coordinate the creation and destruction of component assemblies. Each assembly object represents an assembly instantiation. Assembly objects are created by calling an **AssemblyFactory** object on the host where the assembly object is to be created. The assembly factory is passed a string pointing to the assembly descriptor file.If necessary, the assembly factory brings the assembly descriptor into the local environment and makes its location known to the assembly object.

4. The assembly descriptor uses the assembly descriptor as a recipe for creating the assembly. The descriptor specifies which components and component homes to create, where they are to be located, what components are to be collocated with each other, and what components are to be connected with each other. Based on this information the **Assembly** object creates each component and component home and "hooks-up" the assembly.

5. In creating a component, the **Assembly** object must create a component server, create a container within the server, install a home object within the container, and then use the home to create the component. This work is completed with the help of a set of objects on each host. These are **ServerActivator**, **ComponentServer**, **Container**, and the **ComponentHome**.

Issue – We probably want to call** Container **something different.

6. The **Assembly** object first calls the **ServerActivator** on the target host to create the component server. There is one instance of the **ServerActivator** object on each host. The **Assembly** object creates the component server by calling the **create_component_server** operation on the server activator object. This operation creates an empty server process and returns a reference to the **ComponentServer** object of the newly created process.

Issue – How does the Assembly object find the ServerActivator on the target host? Some kind of naming scheme?

7. Each server contains a single **ComponentServer** object. It is used by the **Assembly** object to create containers within the server. A container is created when the **Assembly** object calls **create_container** on the **ComponentServer** object, passing in a container identifier or list of container attributes. The **create_container** operation returns a reference to the **Container** interface of the newly created container.

8. The **Assembly** object uses the **Container** interface to install the component home into the container. This is accomplished by calling **install_home** on the Container object. The **install_home** operation takes a component **id** parameter and returns a reference to the home interface.

Issue – What about Remote Homes?

9. In order to create the home, the **Container** must load the DLL, shared object file, or .class file into the container process. To determine the path or the or fully qualified name of the component implementation, the container calls the **get_implementation** operation of the **Installation** object. It passes in the **id** of the component implementation and is returned the absolute location or name of the component implementation. The container then loads the implementation and instantiates the home object. The home object reference is then returned to the **Assembly** object.

Note – TODO: Specify DLL, .so, and .class entry points (in separate section).

10. The **Assembly** object uses the component's home object to create the component instance. The instance is created by calling **create_component** on the home reference. **create_component** returns a **ComponentBase** object reference.

Issue – Is there a need to create or find a component with a primary key at deployment time?

11. If applicable, a configurator is applied to the component.

Issue – Are Configurators custom or generic? If custom, then how are they packaged? If generic, then does the property file format have to be specified?

12. Once all of the components are installed, the **Assembly** object connects components in the assembly based on the information in the connect block of the assembly descriptor. It does this by calling the receptacle connect operation on the **ComponentBase** reference.

Issue – How do we connect events? It should be as straight forward as connecting interfaces, and we must be able to connect events on a component basis, not on a container basis.

13. Following the successful consummation of each connection in the assembly, the **Assembly** object calls **configuration_complete** on each object in the assembly to signal that all of its initial connections have been fixed.

9.9.2 Installation Interface

The **Installation** object is used to install, query, and remove component implementations on a single platform. There is at most one **Installation** object per host.

It is intended that this interface be general enough to encompass a wide range of underlying implementations, from ad hoc, to Windows Registry based, to commercial software distribution infrastructure based.

```
exception UnknownImplId { };
exception InvalidLocation { };

interface Installation {
    boolean install(in string implGUID, in string cmpntloc)
        raises InvalidLocation;
    boolean replace(in string implGUID, in string cmpntloc)
        raises InvalidLocation;
    boolean remove(in string implGUID)
        raises UnknownImplId;
    string get_Implementation(in string implGUID)
        raises UnknownImplId;
    string get...(in string key); // TBD
};
```

Issue – On the get... methods we want to be able to get the implementation, such as a DLL or a .class file, as well as other implementation specific dependencies. One possibility is to have get take a string key and return a location for that item. The component factory would have to look in the component descriptor to determine these dependencies and make sure that they are available to the implementation.

Issue – Need operation descriptions

9.9.3 AssemblyFactory Interface

The **AssemblyFactory** interface is used to create **Assembly** objects. A single **AssemblyFactory** object must be present on each host where **Assembly** objects are to be created.

```
exception InvalidLocation { };
exception InvalidAssembly { };

interface AssemblyFactory {
    Cookie create(in string assemblyloc)
        raises InvalidLocation;
    Assembly lookup(in Cookie c)
        raises InvalidAssembly;
    boolean destroy(in Cookie c)
        raises InvalidAssembly;
};
```

Issue – Specify cookies and operation descriptions

9.9.4 Assembly Interface

The **Assembly** interface represents an assembly instantiation. It is used to build up and tear down component assemblies. Building the assembly means that it is going to instantiate all of the components in the assembly and create connections between them as specified in the assembly descriptor. Tearing the assembly down means removing all connections and destroying the components in the assembly.

```
enum AssemblyState {INACTIVE, INSERVICE};

interface Assembly {
    boolean build();
    boolean tear_down();
    AssemblyState get_state();
};
```

Issue – Need operation descriptions

9.9.5 ServerActivator Interface

The **ServerActivator** is a singleton on each host supporting components. It is used to create the servers in which containers and components reside. The **create_component_server** operation returns a **ComponentServer** reference that represents the newly created server process.

```
interface ServerActivator {
    ComponentServer create_component_server();
};
```

Issue – The ServerActivator interface probably belongs in the Container chapter and we need operation descriptions

9.9.6 ComponentServer Interface

There is one **ComponentServer** object per component server process. It is used to create containers within the server.

```
interface ComponentServer {
    Container create_container(...); // params TBD
};
```

Issue – Nail down create_container parameters.

Issue – The ComponentServer interface probably belongs in the Container chapter and we need operation descriptions

9.9.7 Container Interface

The **Container** interface is used to install component homes into a container.

```
interface Container {
    HomeBase install_home();
};
```

Issue – The Container interface probably belongs in the Container chapter. A different name might be in order too.

9.9.8 Component Entry Points (Component Home Factories)

Each component package contains a component implementation. A component implementation is a dynamically loadable module such as a DLL, a shared library, or a Java .class file. The component implementation file contains the code for the component implementation and its associated home implementation.

To load a component into a container, the home for the component must first be created. The home is then used to create component instances. The component's home is created by calling a well known entry point in the component implementation file.

The entry point is an operation or function whose existence and signature is common across all component implementation files. The generic entry point function allows a container to create a component home without having to have specific knowledge of that home or its associated component implementation.

Entry points are programming language specific. Depending on the language, it is either a function or static method. The signature and semantics of the operation are specified for Java and C++.

Entry Points in Java

In Java, the entry point is the name of a class and static method which may be invoked to create a servant which implements the component home. The method must have the following signature:

```
public static HomeExecutorBase
foo();
```

For instance, if one wrote the following code for the entry point:

```
package bigbank.corbacomponents.Account;
public class AccountHomeFactory {
    public static HomeExecutorBase create() {
        return new AccountHomeImpl();
    }
}
```

Then the string representing the entry point string would be "bigbank.corbacomponents.Account.AccountHomeFactory.create".

Entry Points in C++

In C++, the entry point is the symbol in a shared library or DLL which should be invoked to return the **HomeExecutorBase** for the component's home implementation. It should have "C" linkage (i.e. no name-mangling) and have the following signature:

```
HomeExecutorBase* (*)();
```

So for example:

```
extern "C" {
    HomeExecutorBase* createAccountHome() {
        return new AccountHomeImpl();
    };
};
```

In this case, the entry point would simply be "**createAccountHome**".

Component Meta-Model 10

10.1 Introduction

This Chapter provides metamodel representations of the Components Model and Component Descriptors based on the OMG Meta-Object Facility (MOF). The MOF defines a standard means for metamodel definition. It also shows the Document Type Definitions (DTDs) generated from the two metamodels, based upon the XML Metadata Interchange (XMI).

The XMI DTDs and the IDL generated for the metamodels is contained in Appendix C, along with XMI streams that encode the state of the metamodels according to the MOF DTD contained in the XMI specification.

The XMI DTD and MOF-compliant IDL for the CORBA IR metamodel are contained in Appendix C.

10.2 Change History

The following changes have been made since the previous document (orbos/99-02-01) was posted in February:

1. A MOF-base metamodel of the packaging and deployment architecture has been added.
2. The MOF-based metamodel of the Interface Repository has been streamlined resulting in a significantly smaller XMI stream.

The following changes have been made since the previous document (orbos/98-12-02) was posted in December:

1. The MOF descriptive text was greatly reduced.
2. An introduction to XMI was added to understand the DTDs.

3. A MOF-based model for the existing Interface Repository has been added to allow the specification of a model for the component extensions.

4. A MOF-based model for the component IDL extensions has been added based on the prior version of this specification (orbos/98-12-02). It will be updated before the final submission.

5. Miscellaneous clarifications have been made to improve the quality of the text.

All changes are clearly marked with change bars. In general existing text which was moved will not have change bars.

10.3 An Overview of the MOF

Because the OMG Meta Object Facility (MOF) was adopted by the OMG fairly recently, the majority of the OMG membership is not yet familiar with it. Thus, this chapter includes this MOF overview.

The MOF is a generic framework for describing and representing meta-information in an CORBA-based environment. In this context, the term *meta-information* covers any information that in some sense describes other information. This is intended to include such things as:

- Interface definitions for CORBA objects, COM objects, DCE services and so on,
- Service types for the CORBA Trader,
- Meta-data for databases and information retrieval systems,
- Models and project management information for software development tools,
- Mapping descriptions for interoperability tools; e.g. application level bridges.

The MOF is designed to support many different kinds of meta-information. This is achieved by treating the meta-information as information, and formally modeling each distinct kind of meta- information. These formal models are expressed using the meta-modeling constructs provided by the MOF Model[1].

The MOF specification also defines an IDL mapping which allows models expressed using MOF Model constructs to be translated into interfaces – CORBA-based meta-information services. These interfaces can be implemented by hand, or using non-standard server generation tools.

The mapped interfaces for a meta-model all inherit from a standard Reflection module that supports introspection and meta-model independent access and update. The interfaces can be used within a MOF Repository framework, or deployed independently.

10.3.1 The MOF Model

The MOF Model is similar to the concepts of UML[2]. The three kinds of building blocks for a meta-information model are

- **objects** (described by MOF Classes),
- **links** that connect objects (described by MOF Associations), and
- **data values** (described by CORBA IDL types).

Instances of these constructs are organized as MOF Packages.

1. The MOF Model can be thought of as an object modeling language with a standardized abstract syntax and a variety of non-standardized concrete syntaxes or notations.

2. The two were defined in parallel, as a collaborative effort, with the goal of close alignment.

10.3.2 The MOF-IDL Mapping

Simply describing a meta-model using the MOF Model does not mean that a client can store and retrieve the meta-information described by the meta-model. To achieve this in the CORBA context, there need to be a set of CORBA IDL interfaces for a meta-information service, and a server that implements that interface. Furthermore, if meta-information services for a given meta-model are to be interoperable, the corresponding interfaces need to conform to an agreed specification.

To this end, the MOF specification defines a standard mapping from meta-models defined using the MOF Model onto server interfaces. The interfaces themselves are expressed in CORBA IDL that can be generated by instantiating templates defined in the MOF specification. The intended semantics of these interfaces are also defined in the specification.

The MOF IDL mapping rules are defined under the assumption that a “top-level” (i.e. un-nested) Package and its contents are mapped onto CORBA IDL as a single unit. The mapping for MOF Model constructs is as follows:

- **Packages**: each MOF Package maps onto a CORBA IDL module that contains the IDL for all of the Package's contained elements. The module also contains interfaces for Package “instance” (i.e. schema) objects, and for a factory object for these schema objects. The details are as follows:
  - If a Package inherits another Package, there is a corresponding inheritence relationship between the IDL interfaces for the schema objects.
  - If a Package imports another Package, a *#include* statement is inserted at the appropriate point. All references to elements in the imported Package use suitably qualified names.
  - If a Package is nested within another Package, the corresponding modules are also nested.
  - The Package schema interface has IDL attributes that give the object references for the Package's class proxy and association instance objects, and for the schema objects for nested schemas.
- **Classes**: each MOF Class (say *Method*) in a meta-model maps onto two CORBA IDL interfaces:
  - The *Method* interface represents the instances of the MOF Class. This has IDL operations and attributes for each instance-level MOF Attribute and Operation, and each MOF Reference defined for the Class. The *Method* interface inherits *MethodClass*
  - The *MethodClass* interface represents the class proxy for the MOF Class. This has IDL operations and attributes for each classifier level MOF Attribute and Operation defined for the Class. It also has a factory operation for “Foo” instances, and sequence attributes giving all *Method* or *Method* subtype instances created in the Package.
  - The *Method* and *MethodClass* interfaces inherits the interfaces corresponding to the MOF Classes supertypes.

- **Associations**: each MOF Association in a meta-model maps onto a CORBA IDL interface. This defines the operations for querying and updating the links belonging to an association instance. The links are not represented as CORBA object references, but as *struct* data types with two fields.

The various interfaces also contain IDL typedefs, exceptions and constants as required by the Package. The scoping of the IDL reflects the scoping of the meta-model definition.

For more details of the IDL mapping, please refer to the MOF specification.

10.4 An Overview of XMI

The main purpose of the XML Metadata Interchange (XMI) is to enable easy interchange of metadata among modeling tools and OMG MOF based metadata repositories in distributed heterogeneous environments. XMI integrates two key industry standards:

- XML - eXtensible Markup Language, a W3C standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard

The integration of these standards into XMI marries the best of OMG and W3C metadata and modeling technologies, allowing developers of distributed systems to share object models and other metadata over the Internet. XMI, together with MOF and UML form the core of the OMG metadata repository architecture.

XMI was submitted in response to the Stream-based Model Interchange Format RFP, and approved by the Architecture Board in January, 1999. Typically, users of XML define a specific Document Type Definition (DTD) corresponding to a specific category or domain of metadata. XMI, however, is applicable across all categories and domains of metadata. To be interchangeable via XMI, the metadata must have its metamodel defined using the MOF elements.

XMI mainly consists of:

- A set of XML Document Type Definition (DTD) production rules for transforming MOF based metamodels into XML DTDs
- A set of XML Document production rules for encoding and decoding MOF based metadata
- Design principles for XMI based DTDs and XML streams
- Concrete DTDs for UML and MOF

XMI enhances metadata management and metadata interoperability in distributed object environments in general and in distributed development environments in particular. While this response addresses stream based metadata interoperability in the object analysis and design domain, XMI is equally applicable to metadata in many other domains. Examples include metamodels that cover the application development life cycle as well as additional domains such as data warehouse management,

distributed objects and business object management. OMG is expected to issue new RFPs for MOF-compliant metamodels to cover these additional domains.

The adoption of the UML and MOF specifications in 1997 was a key step forward for the OMG and the industry in terms of achieving consensus on modeling technology and repositories after years of failed attempts to unify both areas. The adoption of XMI is expected to reduce the plethora of proprietary metadata interchange formats and minimally successful attempts of the Meta Data Coalition (Meta Data Interchange Specification) and Case Data Interchange Format (EIA CDIF) because of widespread adoption of W3C (XML) and OMG (UML, MOF) standards. XMI is also expected to ease the integration of CORBA, XML, Java, and COM based development environments which are evolving towards similar extensible repository architectures based on standard information models, repository interfaces and interchange formats.

Figure 10-1 below illustrates the manner in which XMI supports interchange. For any MOF-defined metamodel, XMI specifies a DTD. For any model conforming to that metamodel, XMI defines an XML document representing that model. That XML document will conform to the DTD generated from the metamodel.

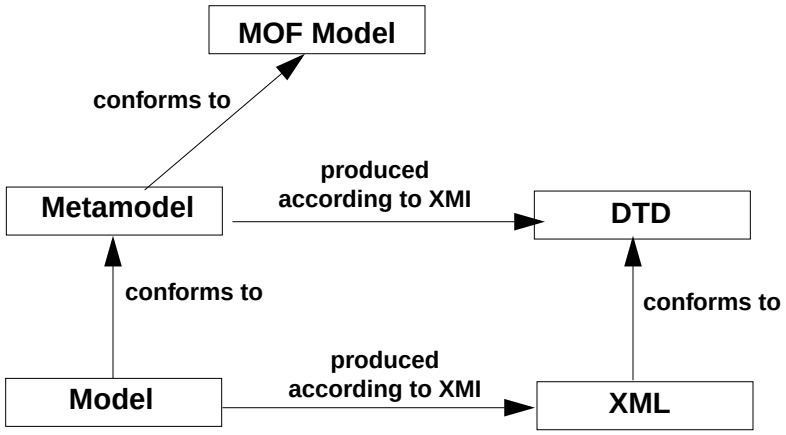

Figure 10-1 Production of Metamodel DTD and Model Streams

10.5 A MOF-Based Interface Repository Metamodel

The first goal of the MOF-compliant metamodel is to express the extensions to IDL defined by the CORBA Component Model. Since these extensions are derived from the previously-existing IDL base, it is not possible to define a MOF-compliant metamodel for the extensions without defining a MOF-compliant metamodel for the IDL base.

Thus, the first MOF Package defined, entitled *BaseIDL*, is a MOF-compliant description of the pre-existing CORBA Interface Repository, while the second Package, entitled *ComponentIDL*, expresses the Component Model extensions. As shown by the following package diagram (Figure 10-2), the *ComponentIDL* Package is dependent upon the *BaseIDL* Package:

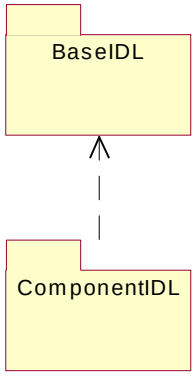

Figure 10-2 The Two Packages for the IDL Metamodel

10.5.1 BaseIDL Package

The base CORBA Interface Repository (IR) is described in the CORBA Core in the form of CORBA IDL. Because the MOF is more expressive than IDL, a range of legitimate MOF-compliant metamodels are equivalent to this IDL. For instance, multi-valued attributes and references expressed in IDL could be ordered or unordered, allow an instance to be contained in the collection only once or more than once, Further, specific multiplicity constraints could be specified. Can the sequence be empty? Is there an upper bound?

As can be seen from an examination of the portion of the metamodel contained in the *BaseIDL* Package, many such questions are resolved via the more precise expression that the MOF enables.

10.5.1.1 A Structural Comparison of the BaseIDL Package with the Existing IR

Although the structure of the MOF-compliant CORBA IR is very similar to the existing CORBA IR, the submitters have taken this opportunity to do some streamlining.

- In the existing CORBA IR, elements that are "typed," such as constants, attributes, etc., hold an attribute of type *IDLType*. However, the same *IDLType* can be the type for many elements, so an attribute (with its composition semantics) is not appropriate. Instead, the MOF-compliant IR specifies the abstract *Typed* metaclass, and an Association between *Typed* and *IDLType*. This change eliminates the need for repeating the *type* attribute, which returns a *TypeCode*, in 6 different metaclasses.
- In the existing CORBA IR, *StructField*, *Parameter*, and *UnionField* are datatypes (structs). The MOF-compliant IR specifies them as full-blown metaclasses so that they can participate as derivations of the *Typed* metaclass.
- The MOF-compliant IR does not have to represent a repository since MOF-based servers inherently have such a construct. Thus, the MOF-compliant IR has no *Repository* metaclass and it specifies *Container* as a sub(meta)class of *Contained*, simplifying the hierarchy.
- The existing IR's *IRObject* provides a *def_kind* readonly attribute. This information would be redundant in a MOF server, which inherently carries information describing the type of a metaobject. Thus, there is no *IRObject* metaclass in the MOF-compliant IR. However, it can be derived for a CORBA IR layer.
- In the existing IR, *UnionDef*, *StructDef*, *ExceptionDef*, and *OperationDef* inherit from *Container*. Since they each contain only a single type of object, it makes less sense for them to have a reference to a collection of *Contained* metaobjects. Instead, in the MOF-compliant IR they each hold their set of fields or parameters as attributes.
- As a simplification the two-stated enums *AttributeMode* and *OperationMode* have been eliminated. Attributes typed as *AttributeMode* or *OperationMode* have been turned into boolean-typed attributes.
- Basic CRUD operations for creating, reading, updating, and deleting metaobjects are generally not included in the metamodel, since these are generated automatically by the MOF-IDL mapping, which takes a MOF-compliant metamodel as input and deterministically derives the IDL for representing the metamodel in a repository.
- The existing IR duplicates many of the interfaces representing basic IR elements with structs representing the same elements. This duplication supports the ability to get a large collection of information required by a DII client without requiring the client to subsequently make repeated, possibly remote requests to objects in order to process the collection of information. Since the DII is optimized for the existing IR, this submission assumes that an IR layer will continue to service DII clients and thus does not attempt to provide this functionality in the MOF-compliant IR.

Figure 10-3 shows all of the metaclasses and relationships defined in the *BaseIDL* Package.

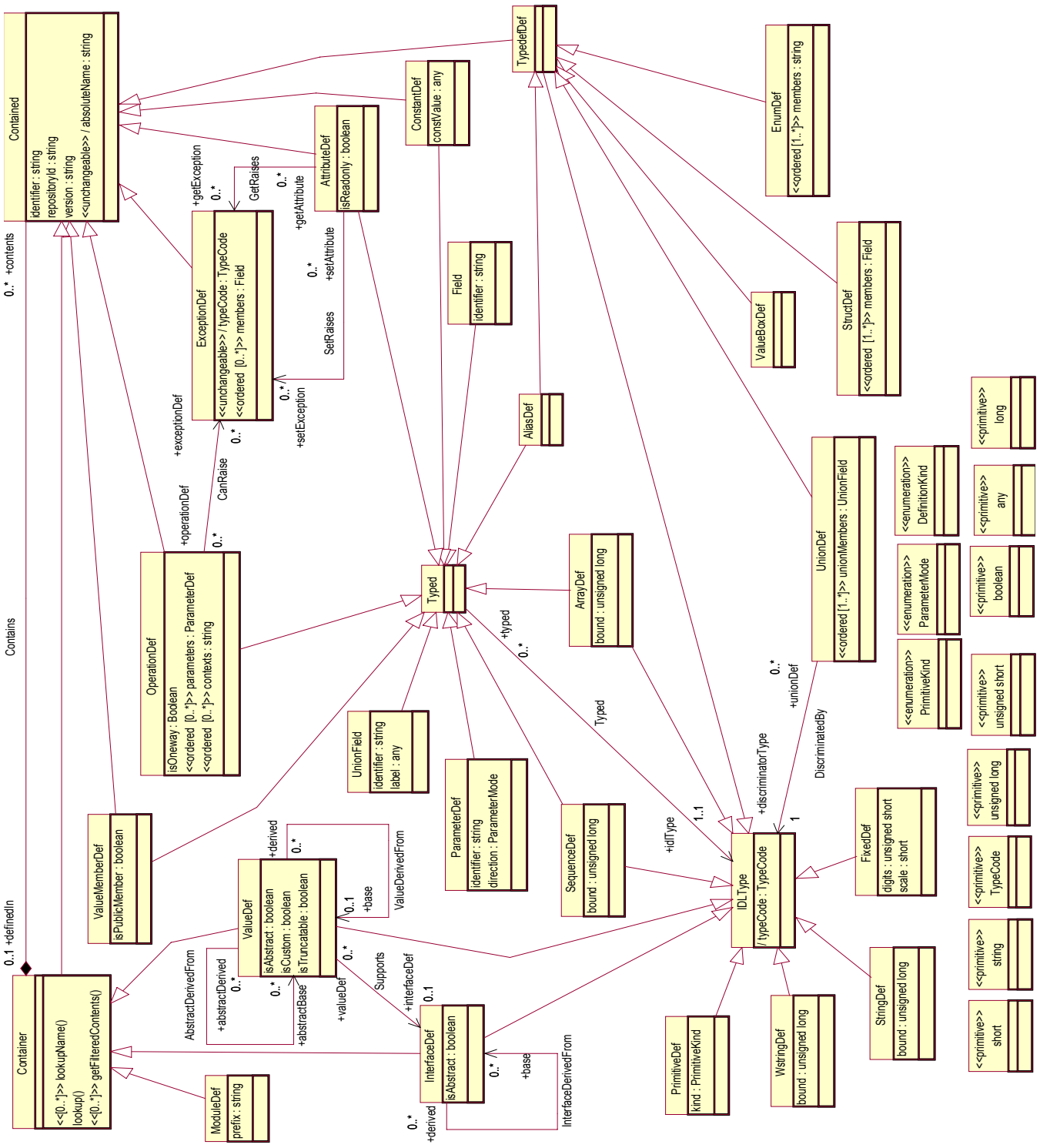

Figure 10-3 BaseIDL Package--All Elements

10.5.1.2 Typing

As mentioned earlier in this chapter (A Structural Comparison of the BaseIDL Package with the Existing IR on page 298), the two critical elements of the BaseIDL Package supporting the typing of IR entities are the *Typed* and *IDLType* metaclasses. A *Typed* element references an *IDLType*, which has an attribute of type *TypeCode.*

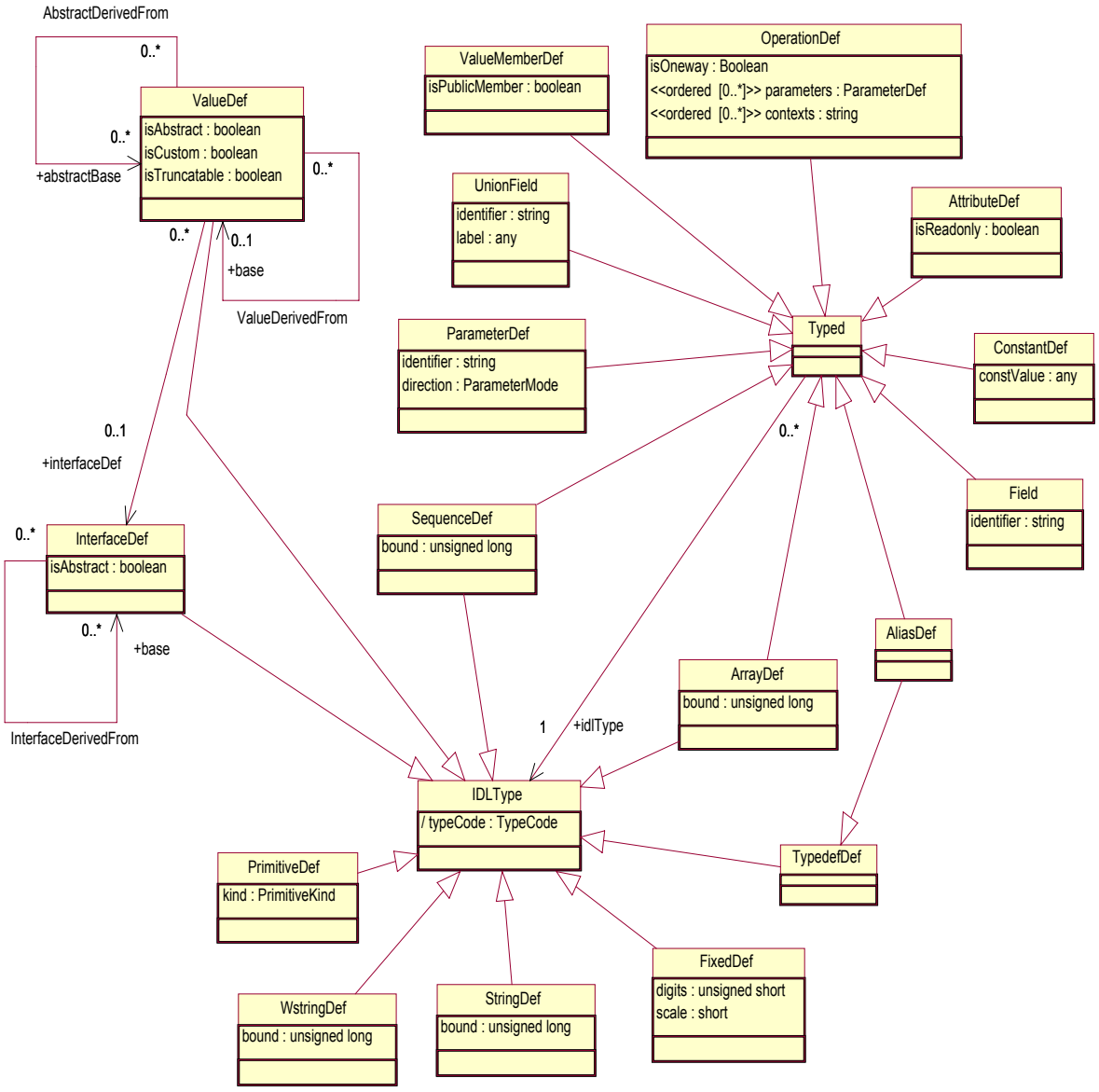

Figure 10-4 IDL Typing

10.5.1.3 Containment

Many elements in the metamodel descend from *Container* or *Contained*, in keeping with the structure of the original CORBA Interface Repository. As mentioned in the previous section, the metamodel also derives *Container* from *Contained* so that an element that is logically a container and at the same time is defined in another container does not have to inherit directly from both *Container* and *Contained*. However, this change requires that a constraint be written such that *ModuleDef* and only *ModuleDef* does not have to be defined in a *Container*. This constraint is included in the next section on containment constraints.

Figure 10-5 expresses the containment hierarchy.

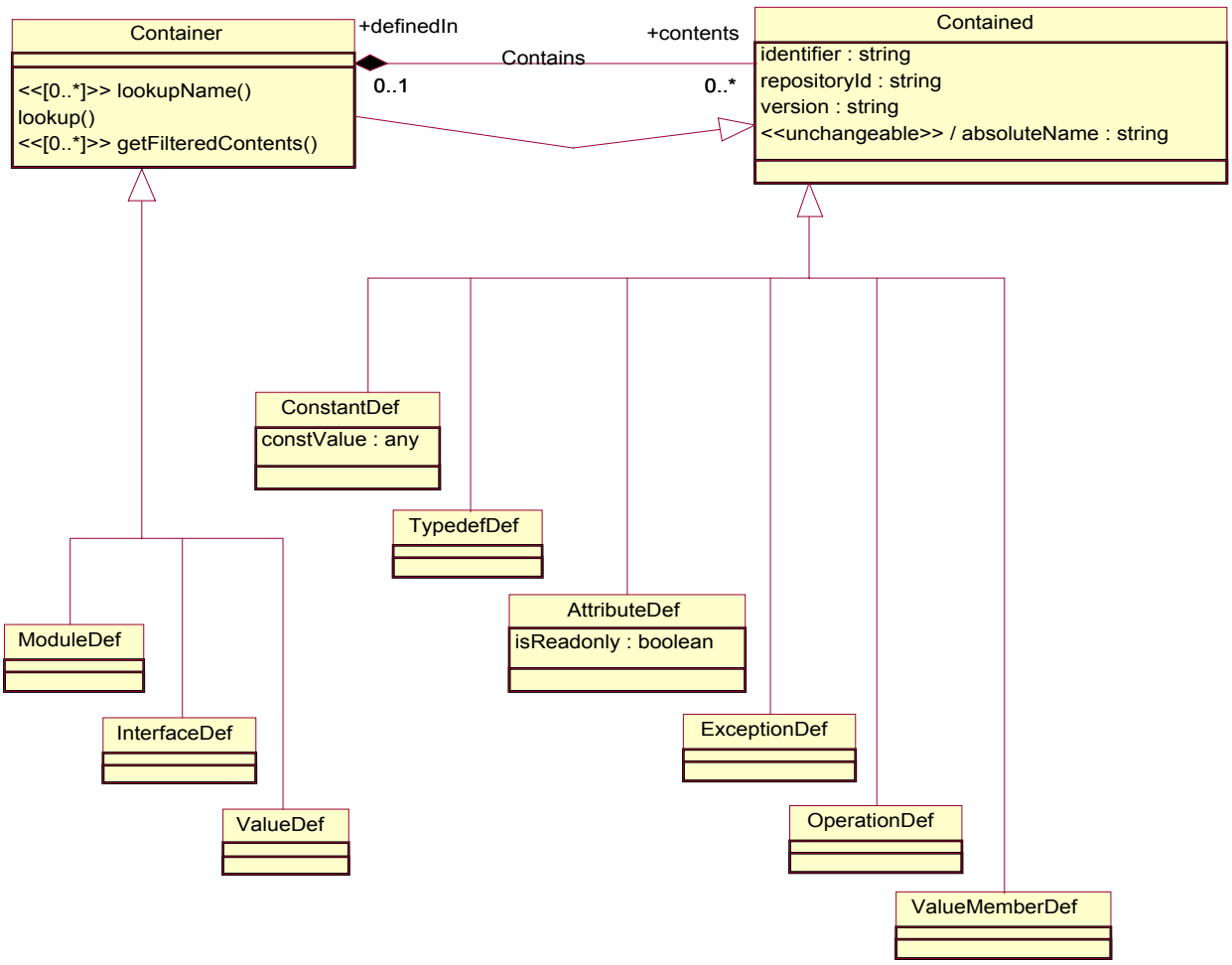

Figure 10-5 Containment Hierarchy

10.5.1.4 Containment Constraints

The Association between *Container* and *Contained* is named *Contains*. *Contains* is very general and is inherited by sub(meta)classes of *Container* and *Contained.* Unless further constrained, *Contains* would allow any *Container* to directly contain any *Contained* element. For example, a *ModuleDef* could contain an *OperationDef* and a *ValueDef* could contain an *InterfaceDef*. Clearly, the *Contains* Association must be constrained.

Figure 10-6 and Figure 10-7 express the containment constraints formally via the OMG's Object Constraint Language (OCL). They also supplement the formal expressions with English natural language equivalents.

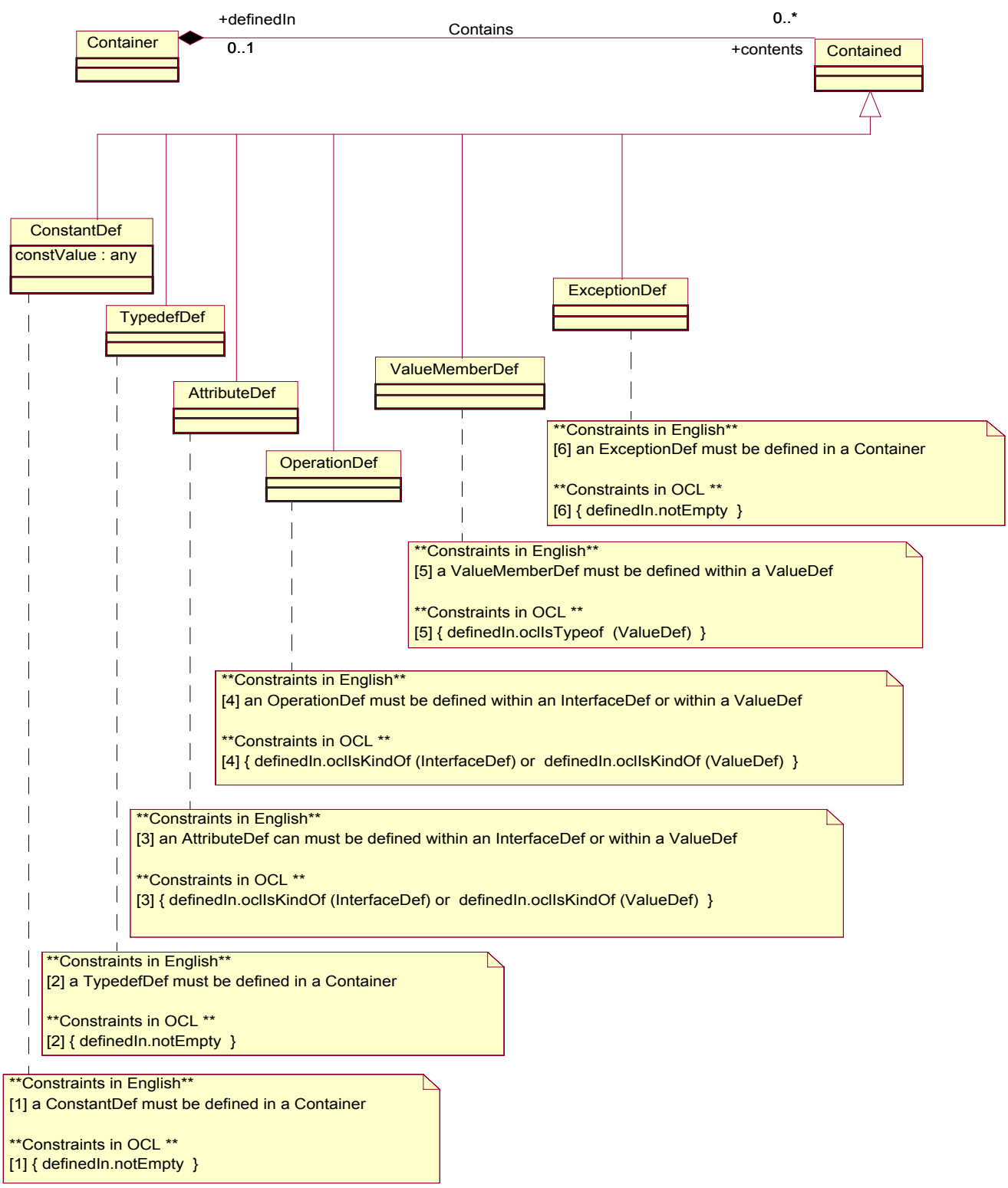

Figure 10-6 Containment Constraints--Subclasses of *Contained*

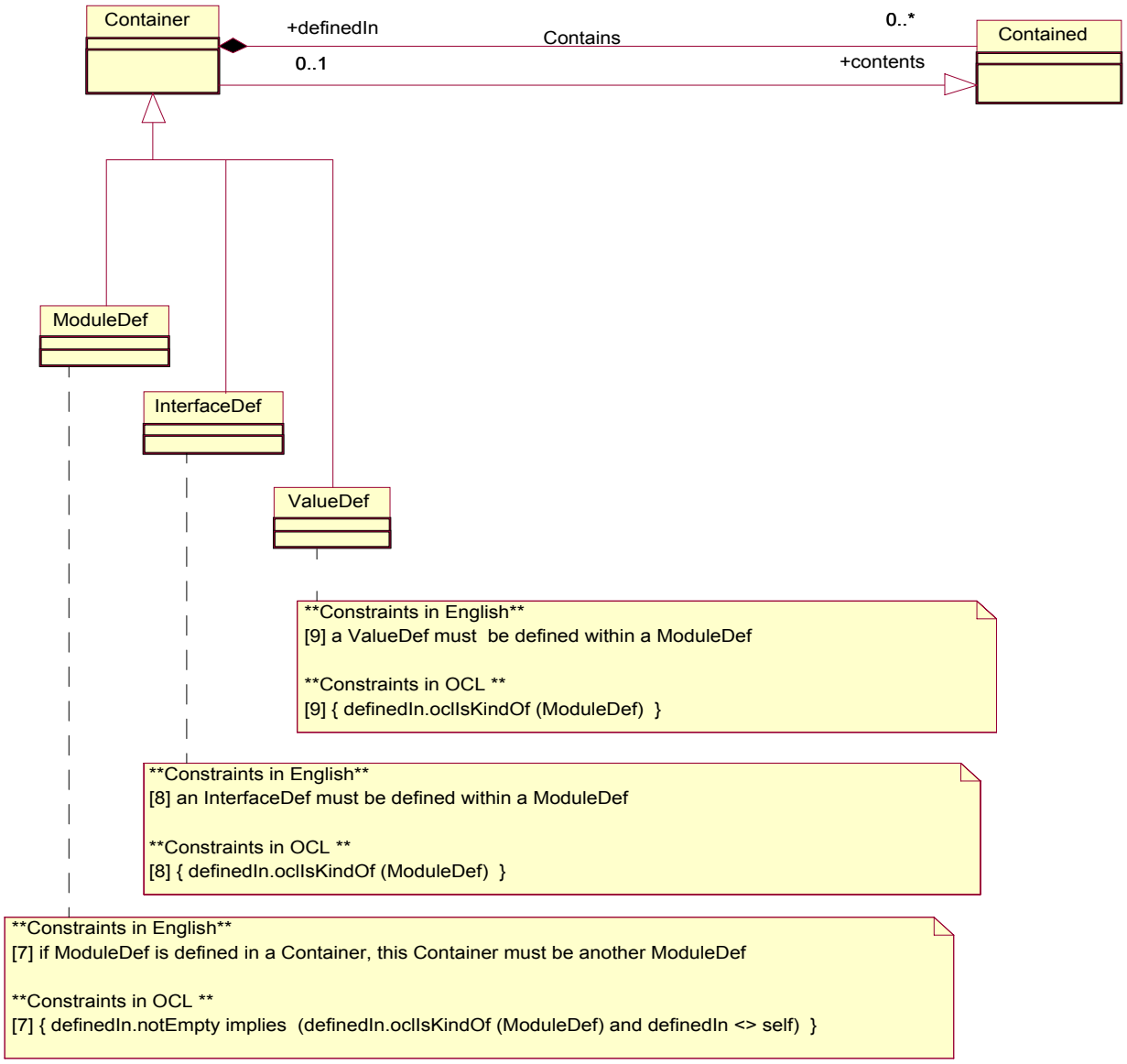

Figure 10-7 Containment Constraints--Subclasses of *Container*

10.5.1.5 Typedef and Type Derivations

Figure 10-8 expresses the hierarchy of derivatives of *Typedef* and *Typed*.

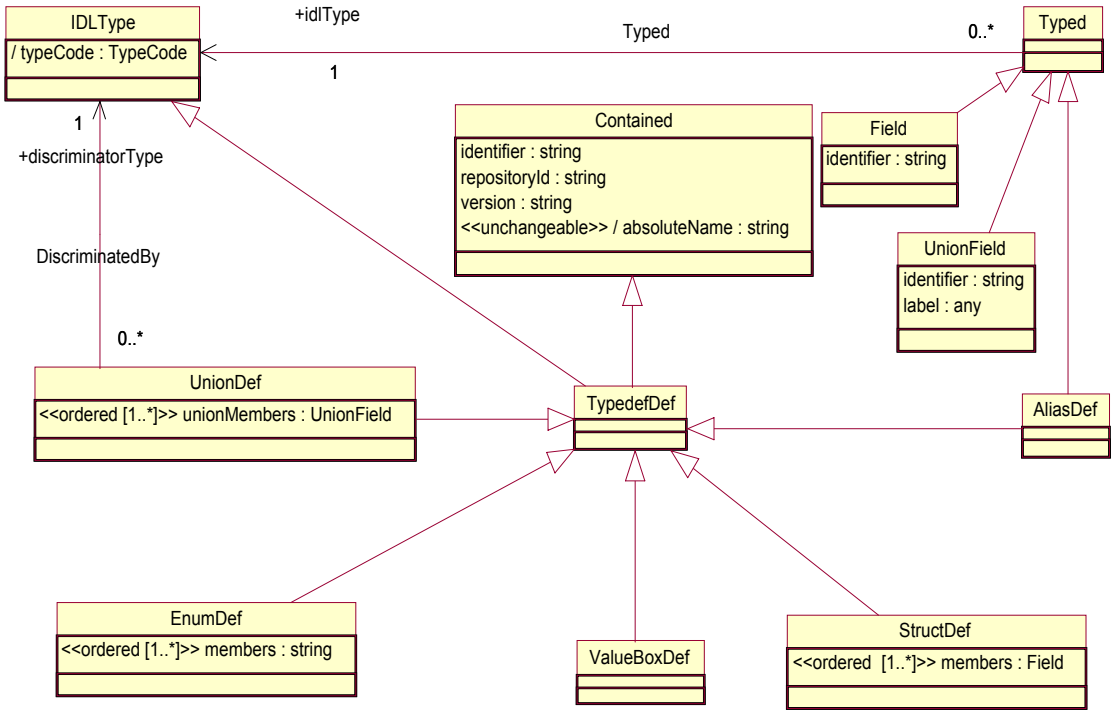

Figure 10-8 Derivations from *Typedef* and *Type*

10.5.1.6 Exceptions

Figure 10-9 shows the formal definition of the *ExceptionDef* metaclass. Note the inclusion of the newly-defined (in this submission) ability for attribute accessors and mutators to raise user-defined exceptions.

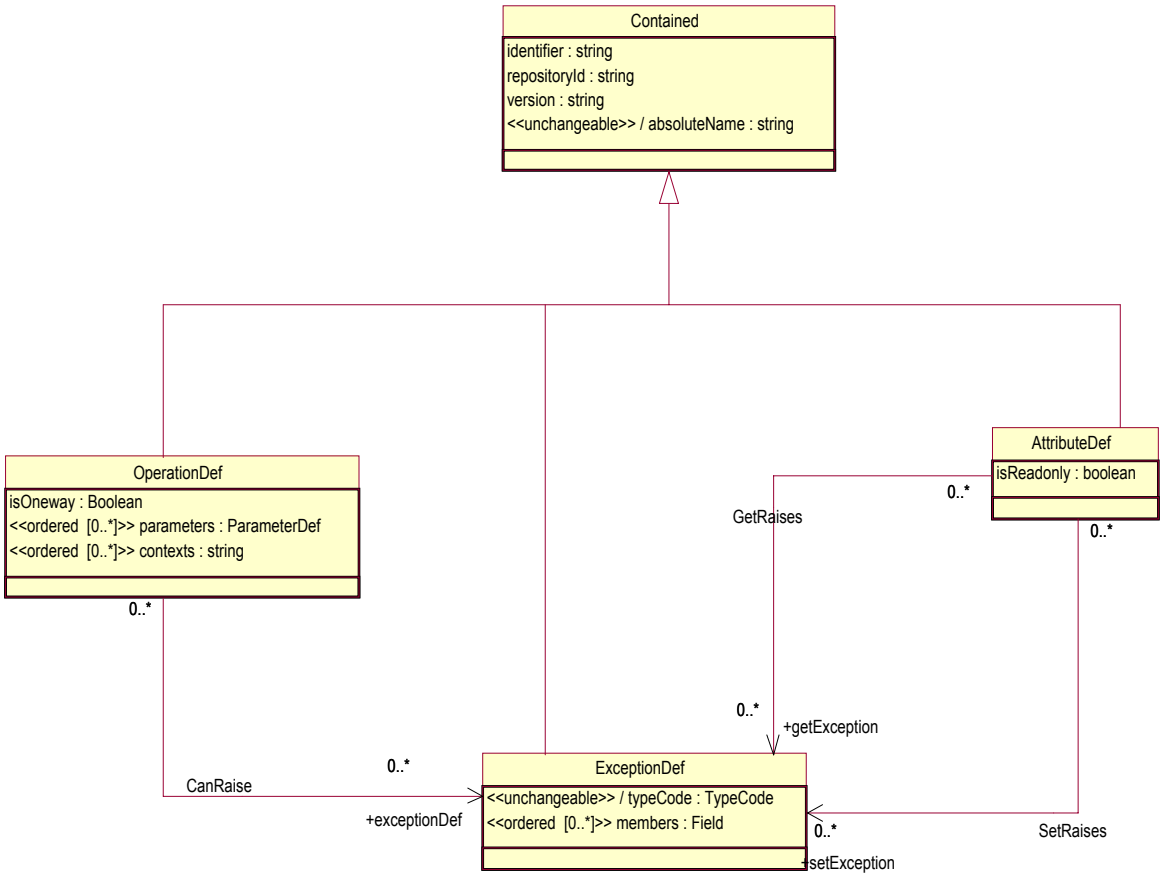

Figure 10-9 Exceptions

10.5.1.7 Value Types

CORBA 2.3 provided a model for types of objects that can be passed by value. The Objects By Value specification expanded the grammar of IDL and the structure of the Interface Repository to accommodate value types. Figure 10-10 focuses on the definition of value types in the MOF-compliant IR metamodel.

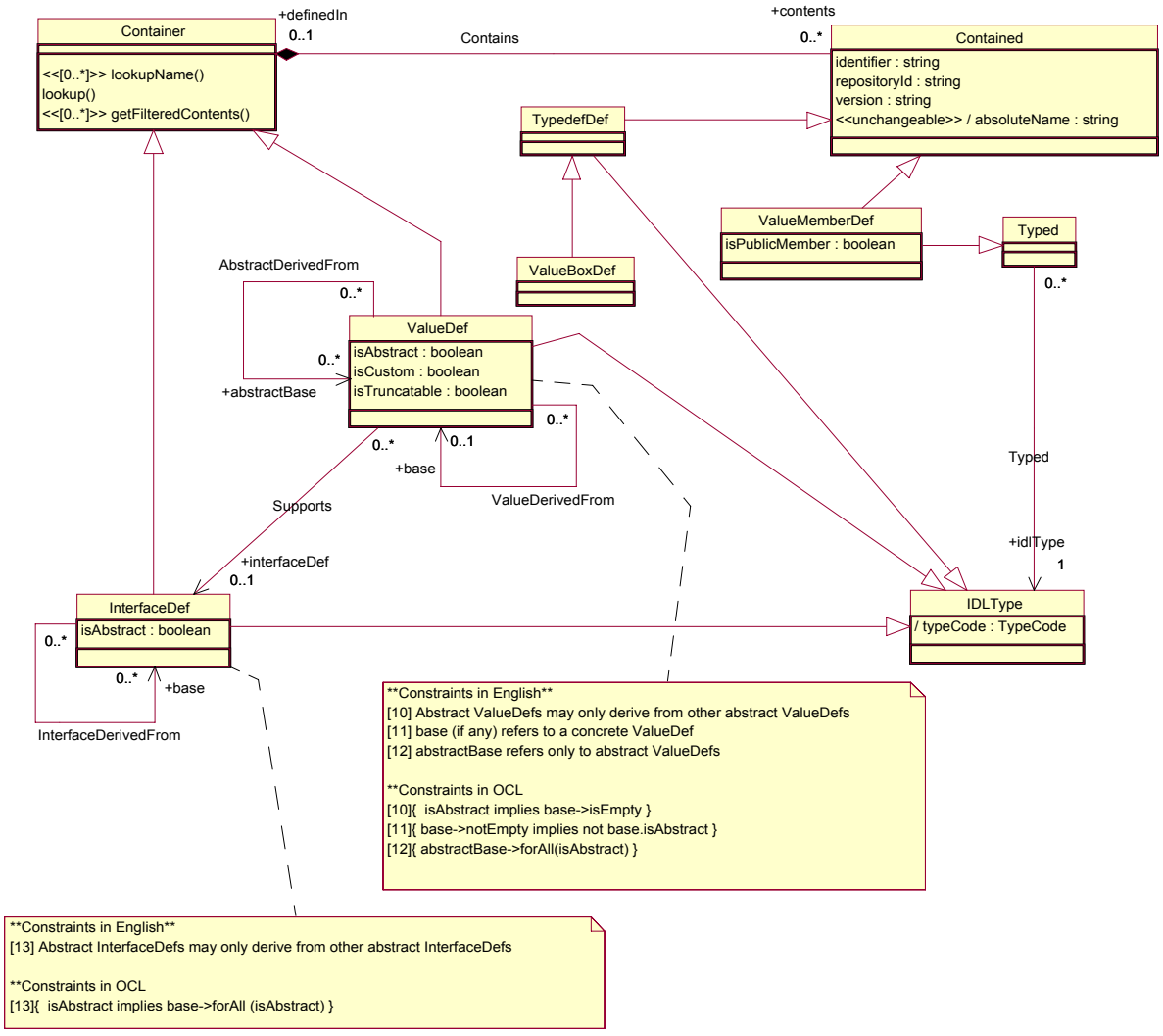

Figure 10-10 Value Types

10.5.1.8 Naming

Figure 10-11 focuses on the aspects of the metamodel that concern naming.

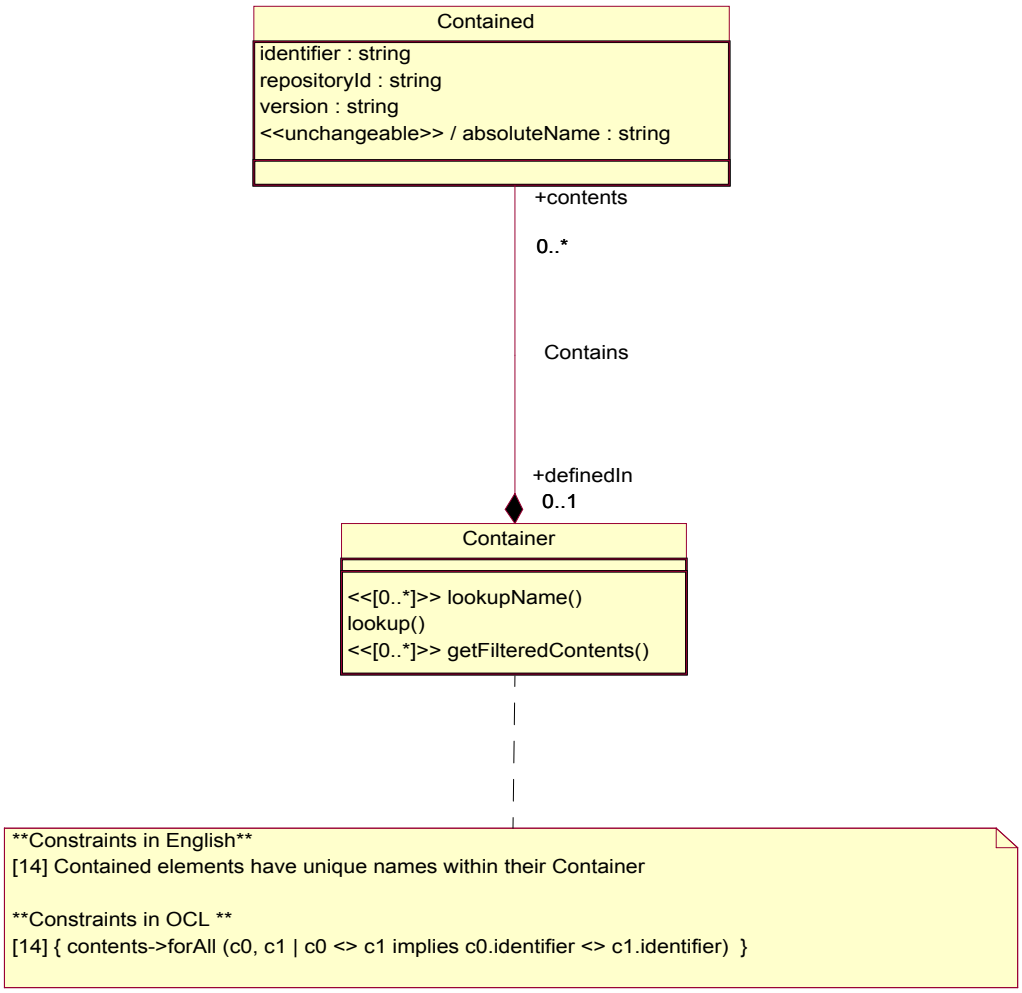

Figure 10-11 Naming

10.5.1.9 Operations

As mentioned earlier in this chapter (A Structural Comparison of the BaseIDL Package with the Existing IR on page 298), the metamodel generally does not declare CRUD operations for the metaclasses, due to the fact that the MOF automatically generates such operations based on the structural metamodel. However, a few convenience operations are defined on the *Container* metaclass, as illustrated by Figure 10-12.

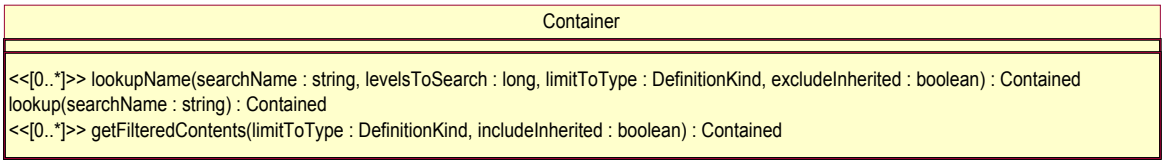

Figure 10-12 Convenience Operations

10.5.2 ComponentIDL Package

10.5.2.1 Overview

The following UML class diagram describes a metamodel representing the extensions to IDL defined by the CORBA Component Model. Just as these extensions are dependent on the base IDL defined in the CORBA Core, so is this metamodel dependent on a metamodel representing the base IDL.

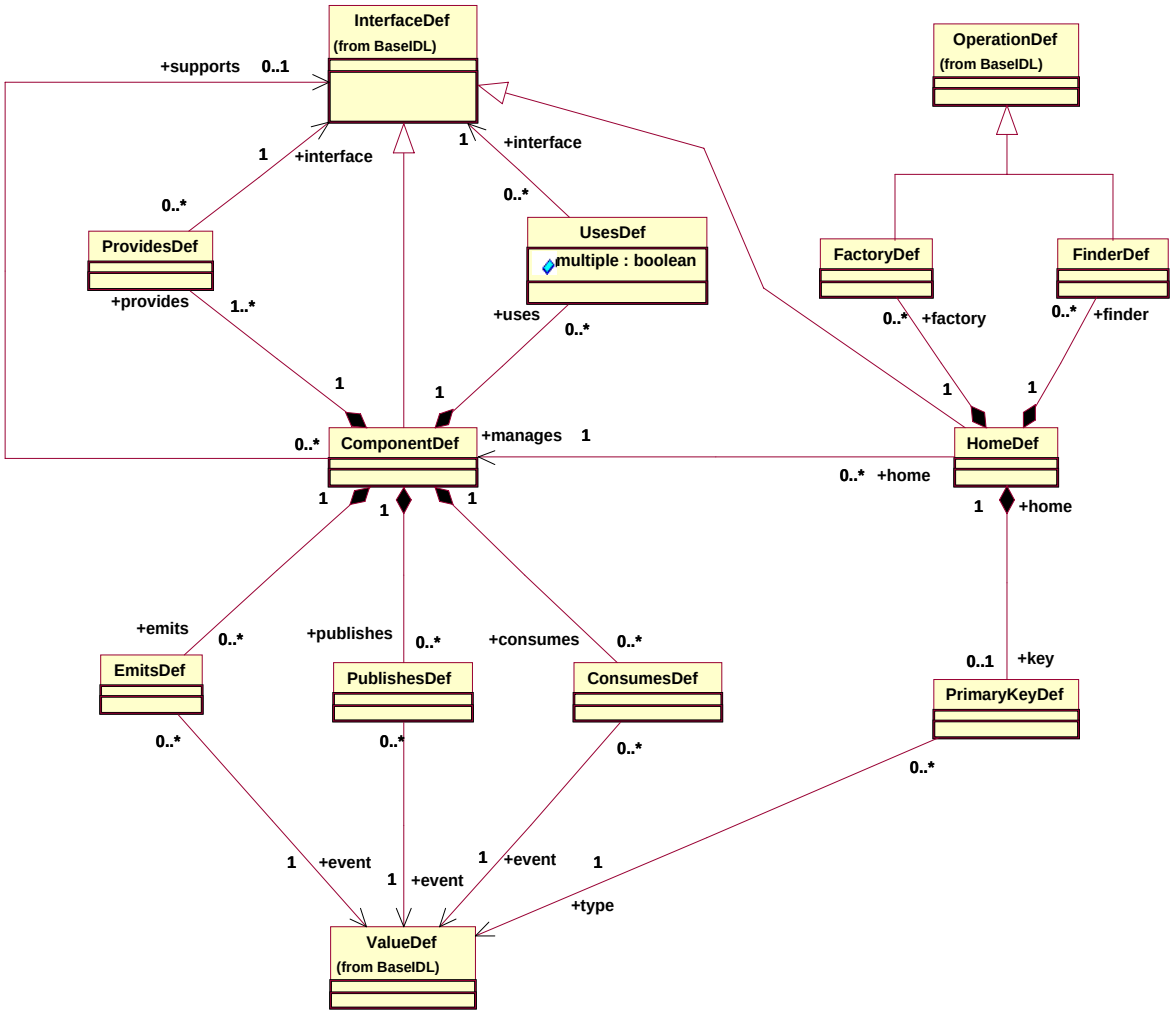

Figure 10-13 ComponentIDL Package - Main Diagram

10.5.2.2 Containers and Contained Elements

The following UML class diagram (Figure 10-14) describes the derivation of the metamodel elements from the BaseIDL *Container* and *Contained* elements:

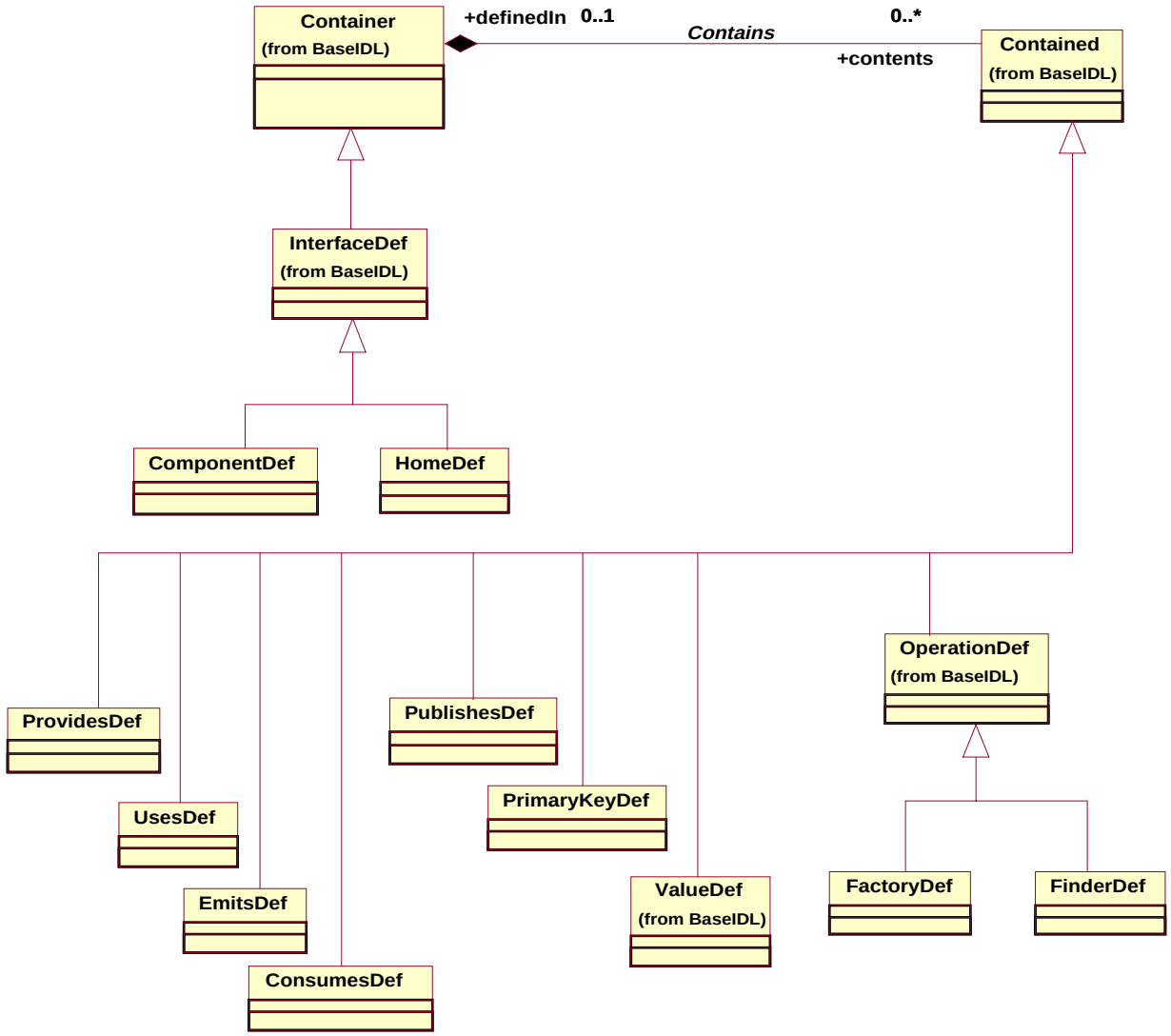

Figure 10-14 Containment Hierarchy

Each of the subtypes of *Contained* shown in Figure 10-14 can only be defined within certain subtypes of *Container*. Figure 10-15 formally specifies these constraints via the OMG's Object Constraint Language (OCL), and supplements the OCL by expressing the constraints in natural language for the benefit of readers who are not familiar with OCL.

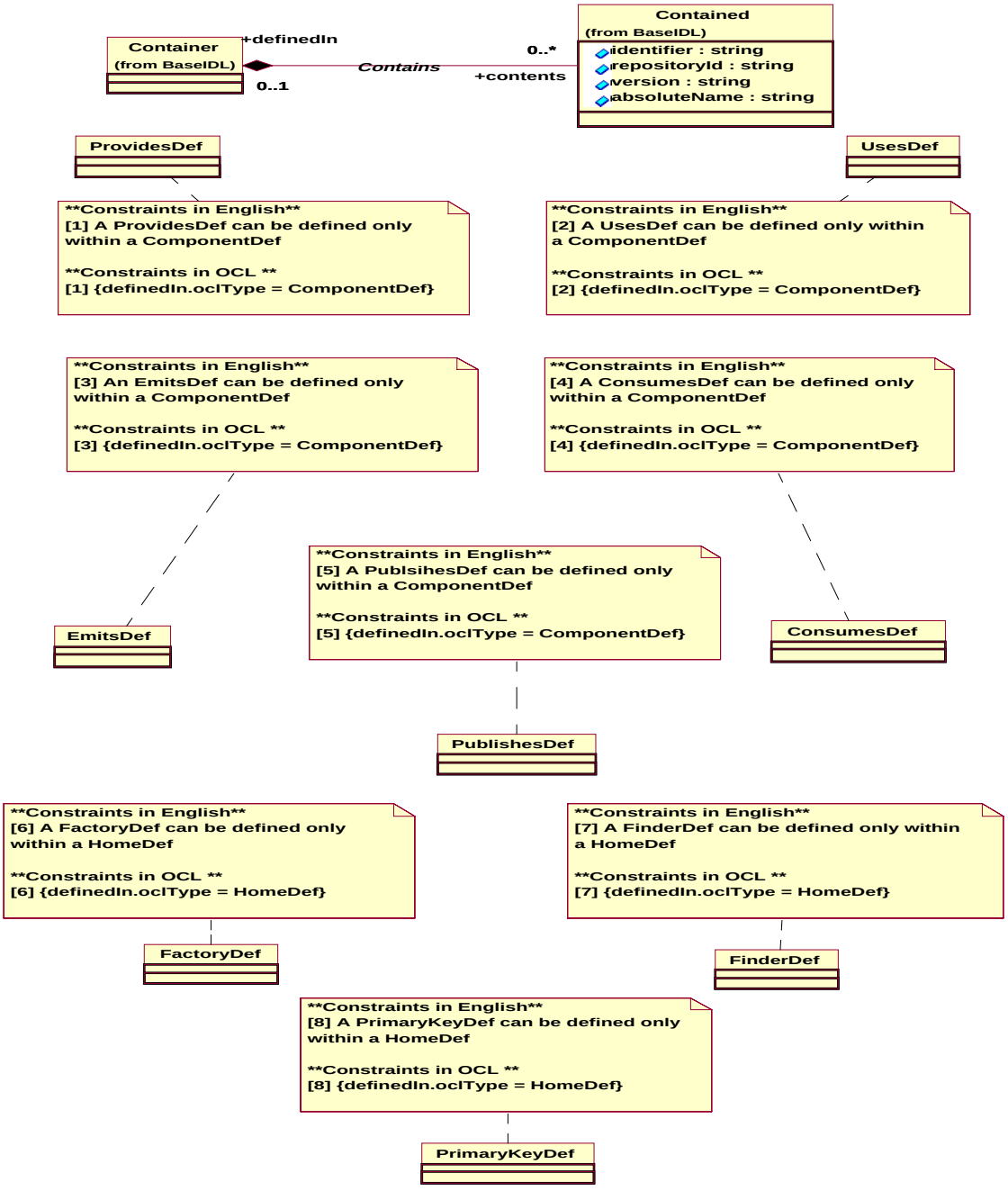

Figure 10-15 Constraints on Containment of Elements Defined In ComponentDef

All of *ComponentDef*'s composition Associations shown in the main diagram (Figure 10-13) are derived from the *BaseIDL* metamodel's *Contains* Association between *Container* and *Contained*. As shown by Figure 10-14, *ComponentDef* inherits that Association from *InterfaceDef*, which inherits it from *Container*.

The following class diagram (Figure 10-16) details these derived Associations. A "/" prefix in an Association name denotes that the Association is derived, and sets the MOF's "isDerived" property for the Association. The constraints for each of the derived Associations are expressed in the OMG's Object Constraint Language and declare how the Associations are derived from the *Contains* Association.

The <<implicit>> stereotype is a standard UML stereotype that designates the Association as conceptual rather than manifest. An <<implicit>> Association is ignored when generating IDL for the metamodel via the MOF-IDL mapping. It is also ignored when deriving the XML DTD for the metamodel via the MOF-XML mapping specified by the XMI specification. The *Contains* association is sufficient for generating the accessor methods in the IDL allowing the containments to be traversed. If these Associations were not marked as <<implicit>> then additional accessor methods would be generated to do the more focused traversals that they conceptualize. In the judgement of the submitters the generation of these additional accessor methods would expand the footprint of the IDL interfaces more than is warranted, given that the containments can be traversed by the single inherited *Contains* Association.

The fact that these <<implicit>> Associations are ignored when generating the IDL for the metamodel does not mean that they have no bearing on the contents of a repository. The "reflective" interfaces that all MOF metaobjects inherit have an operation called *metaObject* that returns a metaobject. This metaobject is part of the metamodel rather than part of a model; in other words, it is actually a meta-metaobject that is part of the description of the metamodel. The definitions of the <<implicit>> Associations in which a metaobject participates would be available via this meta-metaobject. The multiplicity constraints of these Associations would be available as well. Thus, for example, the fact that a *ComponentDef* aggregates zero or more *UsesDef* metaobjects is discoverable through such meta-metaobjects and thus serves as a formal constraint on the *Contains* Association from which the aggregation is derived.

Furthermore, when the state of the metamodel is streamed in conformance with the DTD for the MOF meta-metamodel, the state that specifies the <<implicit>> Associations are part of the stream. The DTD for the MOF meta-metamodel is contained in the XMI specification. XML streams conforming to that DTD and which contain the state of the IR metamodel are included in Appendix C of this submission.

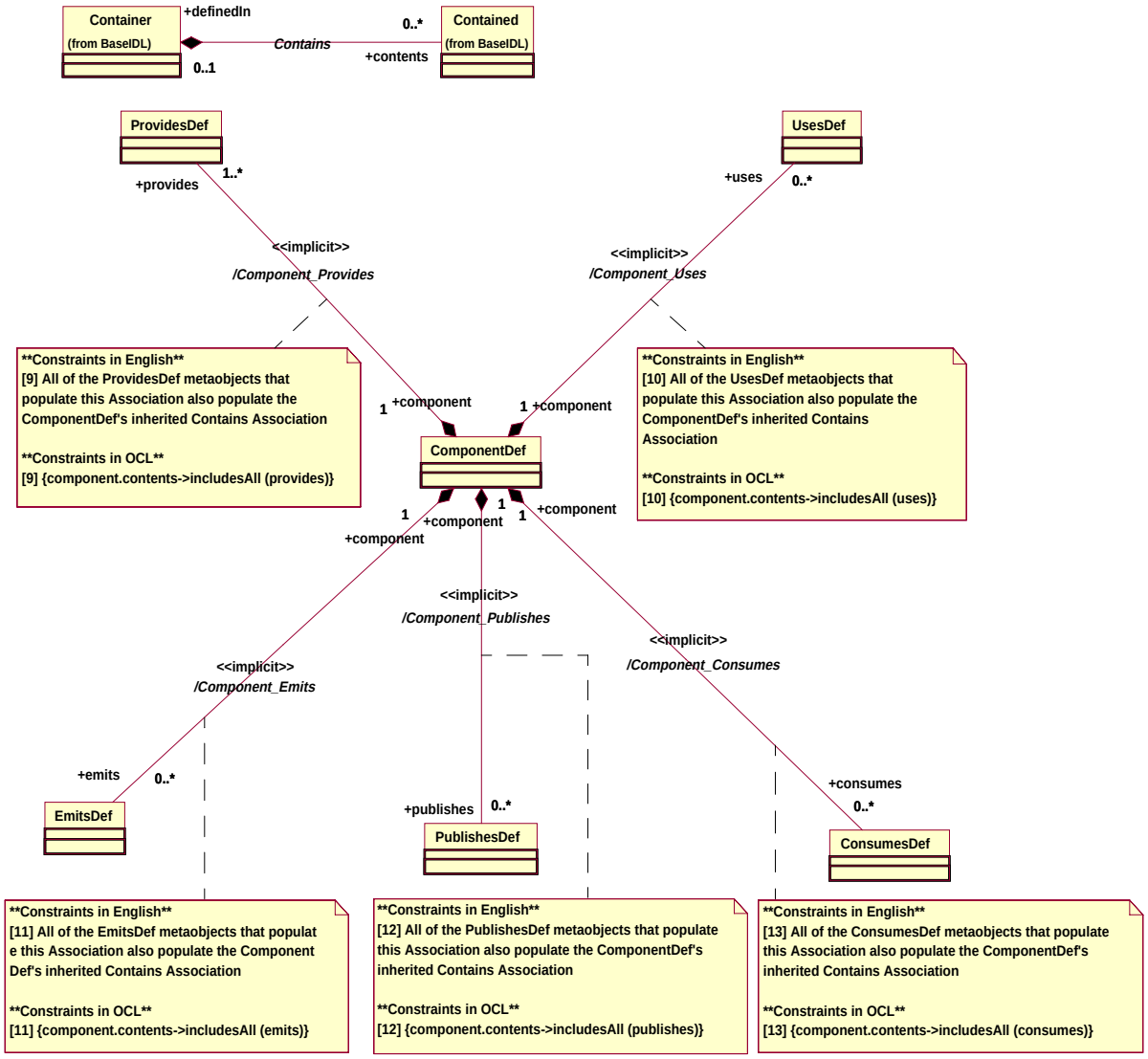

Figure 10-16 Implicit Derived Containments with ComponentDef as the Composite

HomeDef's composition Associations also are derived from the *Contains* Association. As shown in Figure 10-14, *HomeDef* descends from *Container*. All of the components of its composition Associations descend from *Contained*. As with the derived Associations in which *ComponentDef* plays the composite role, the derived Associations in which *HomeDef* plays the composite role are marked as <<implicit>> to prevent excess IDL generation. Figure 10-17 formally defines the constraints that define the semantics of the derivations.

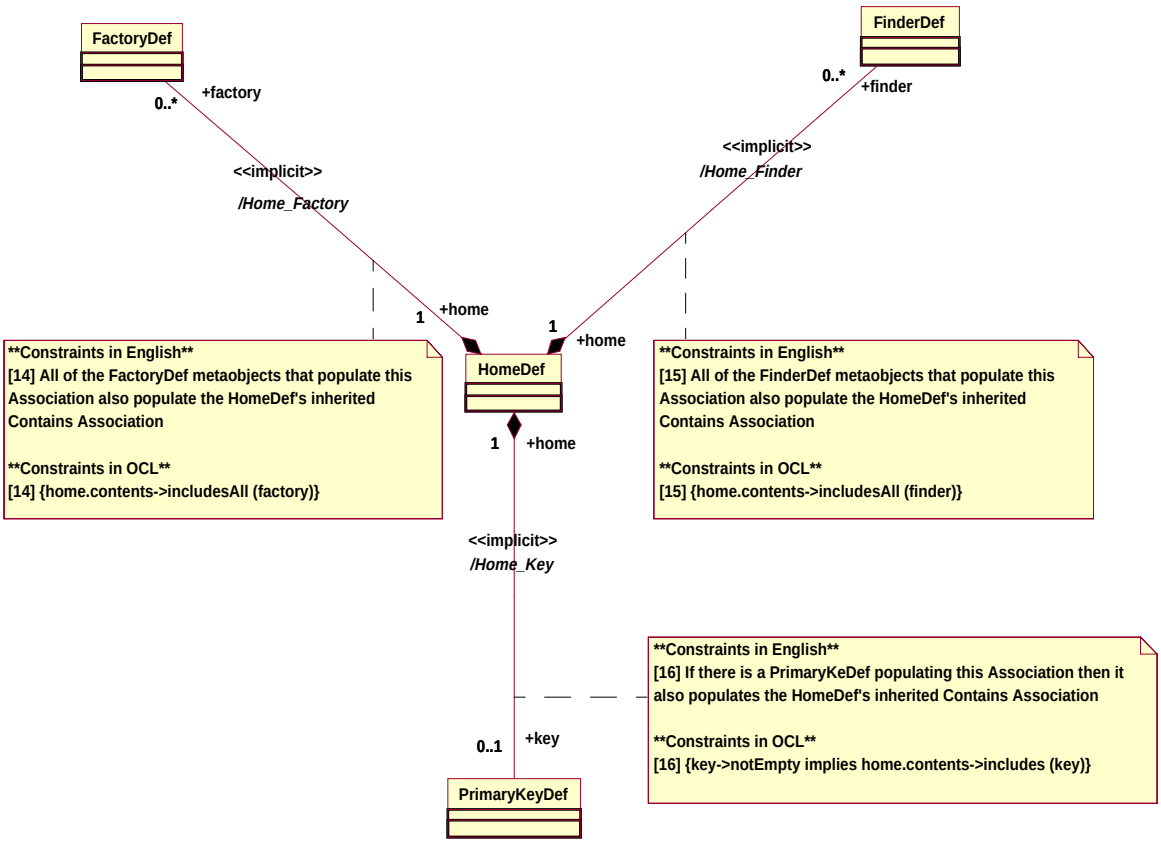

Figure 10-17 Implicit Derived Containments with HomeDef as the Composite

10.5.2.3 ValueDef Constraints

The *ValueDef* metaclass, which is part of the *BaseIDL* Package, participates in a number of Associations defined by the *ComponentIDL* Package. The *emits*, *publishes*, and *consumes* declarations that are part of the component model IDL extensions all reference a *ValueDef*. Furthermore, the *primaryKey* declaration within *home* declarations references a *ValueDef*. However, the IDL type of the *ValueDef* is constrained, as explained in Section 6.8.4 on 119 of this submission.

Figure 10-18 expresses the *ValueDef* constraints formally. Note that it uses an OCL technique of defining a side-effect free operation in order to support recursion, which is required in order to traverse the transitive closure of a *ValueDef's* inheritance hierarchy.

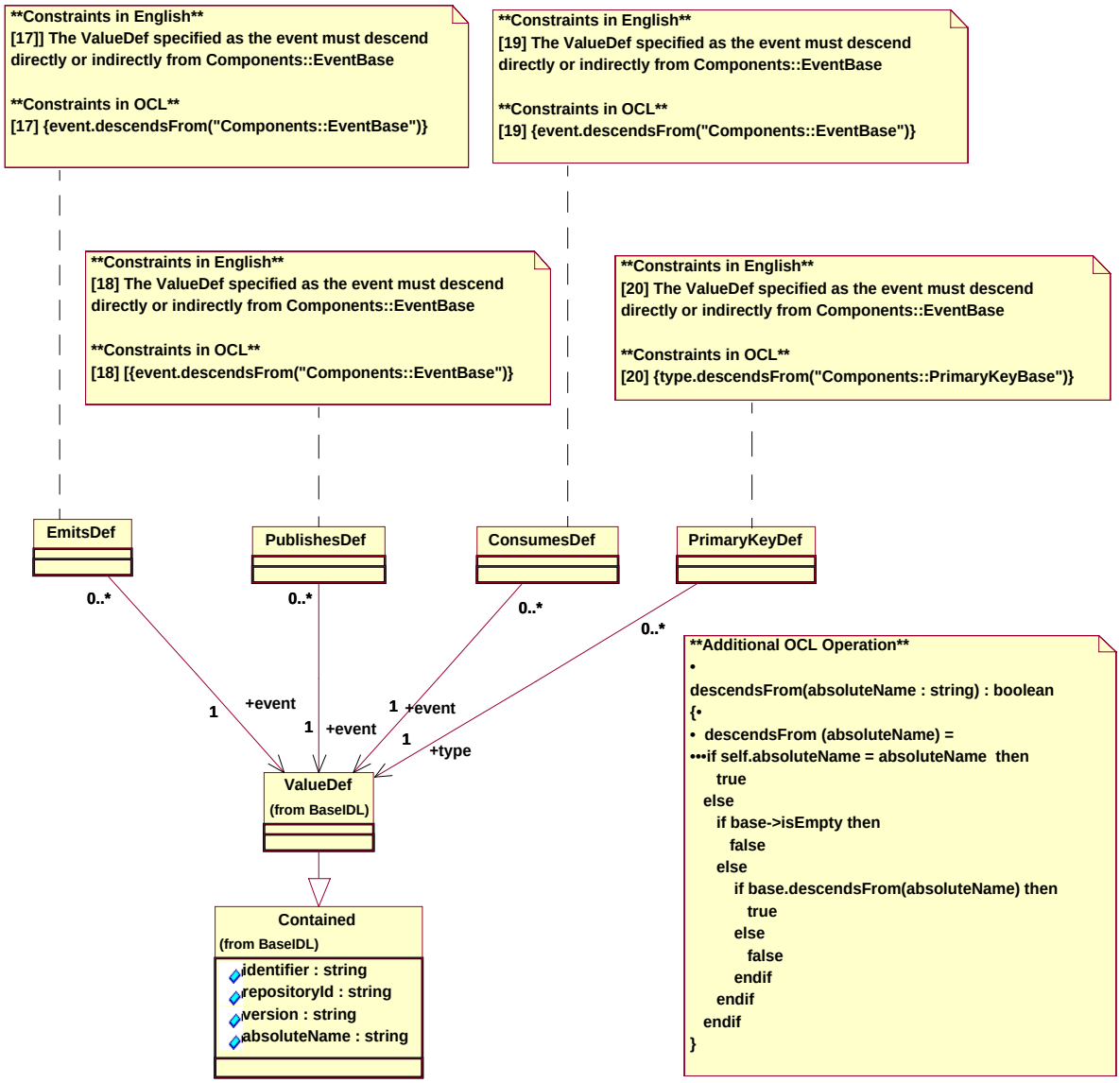

Figure 10-18 Constraints on ValueDefs in Associations

10.5.2.4 Miscellaneous Constraints

Figure 10-19 defines additional constraints on the *ComponentDef*, *HomeDef*, *FactoryDef*, and *FinderDef* metaclasses.

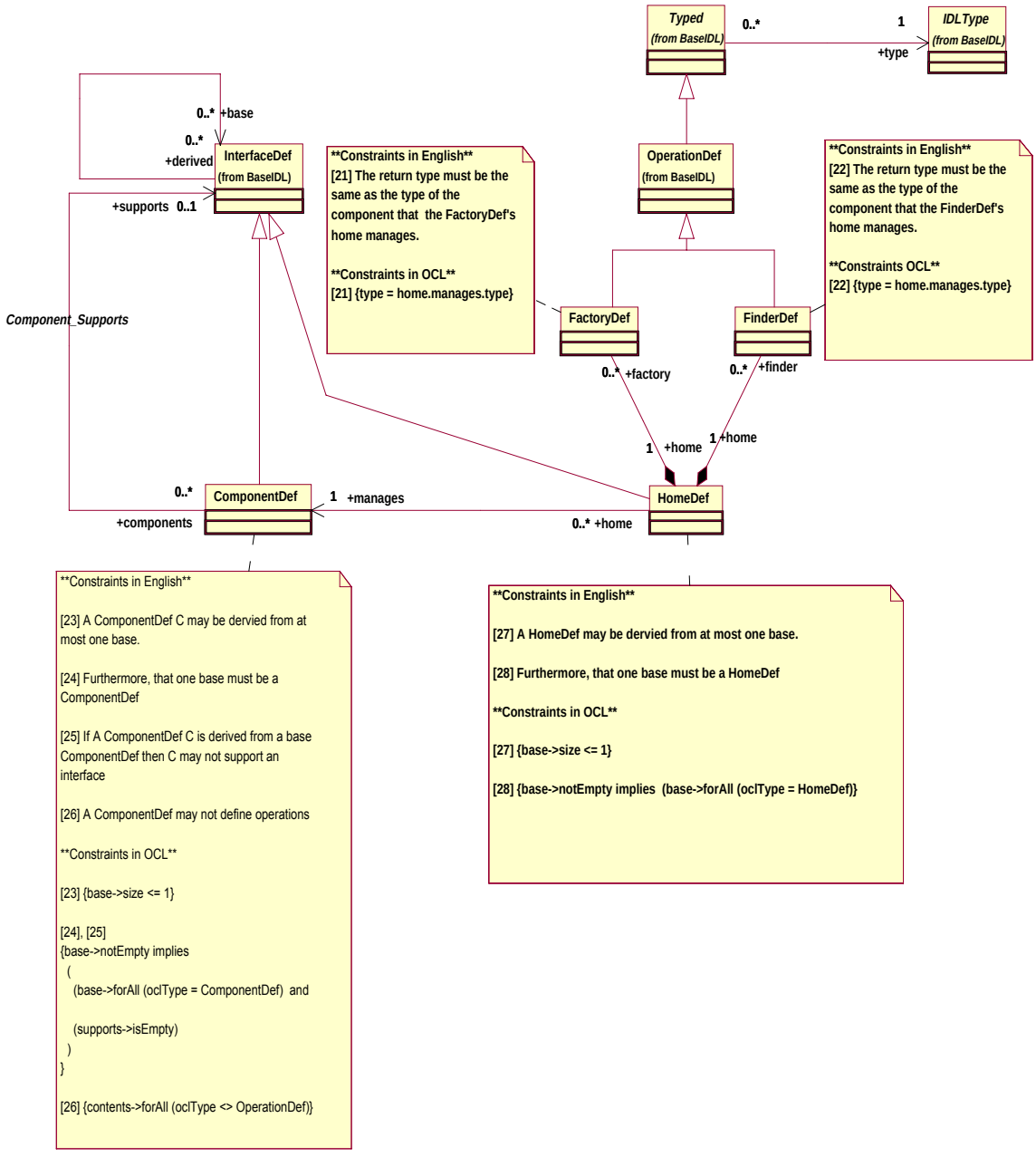

Figure 10-19 Miscellaneous Constraints

10.6 Packaging and Deployment Metamodel

Chapter 9 describes the semantics of metadata “descriptors” describing the packaging and deployment of CORBA components. This section presents a MOF-compliant metamodel of the same information, allowing an XMI-compliant DTD to be generated along with IDL representing an active, message-based, MOF-compliant repository for this metadata.

The metamodel is comprised of five MOF Packages, as illustrated by Figure 10-20. These packages correspond to the organization of the descriptors in Chapter 9.

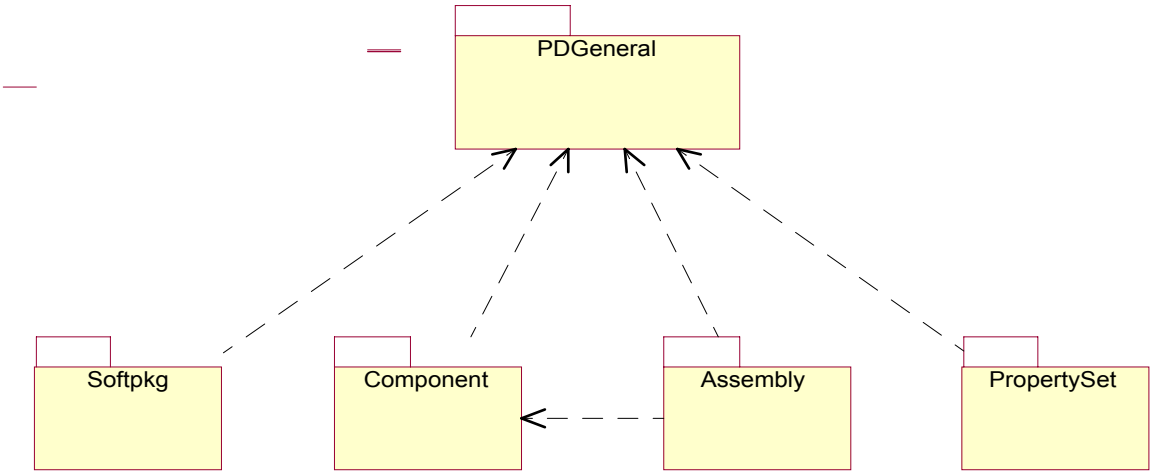

Figure 10-20 MOF Packages Comprising the Packaging and Deployment Metamodel

- *PDGeneral* contains general elements used by the other Packages.
- *Softpkg* concerns the packaging of component implementations.
- *Component* contains a subset of the information about a component that is contained in the Interface Repository, in order to support configuration and deployment without necessitating accessing the Interface Repository.
- *Assembly* provides information required to plug components together and coordinate their deployment.
- *PropertySet* models the contents of a properties file used to hold component configurations.

The metamodel reflects the XML contained in the Packaging and Deployment chapter closely, except that in a number of cases XML DTD Elements were expressed as attributes in the metamodel for efficiency. The metamodel also does some consolidation via inheritance, whereas XML does not support inheritance.

10.6.1 The PDGeneral MOF Package

This Package corresponds to the “General Purpose Elements and Entities” described in Chapter 9. However, the *Repository* element was also placed in this Package because it is used by both the *Softpkg* and *Component* Packages.

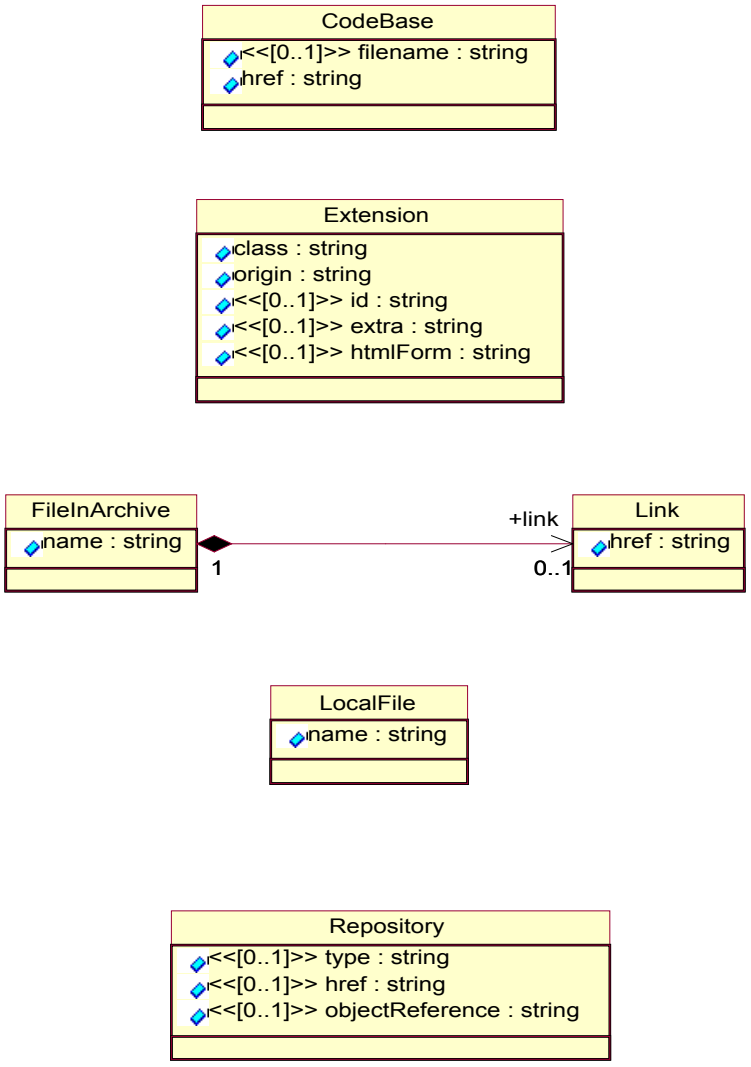

Figure 10-21 General Elements

10.6.2 The Softpkg MOF Package

This package corresponds to the *Softpkg Descriptor* described in Section 9.3 on 230.

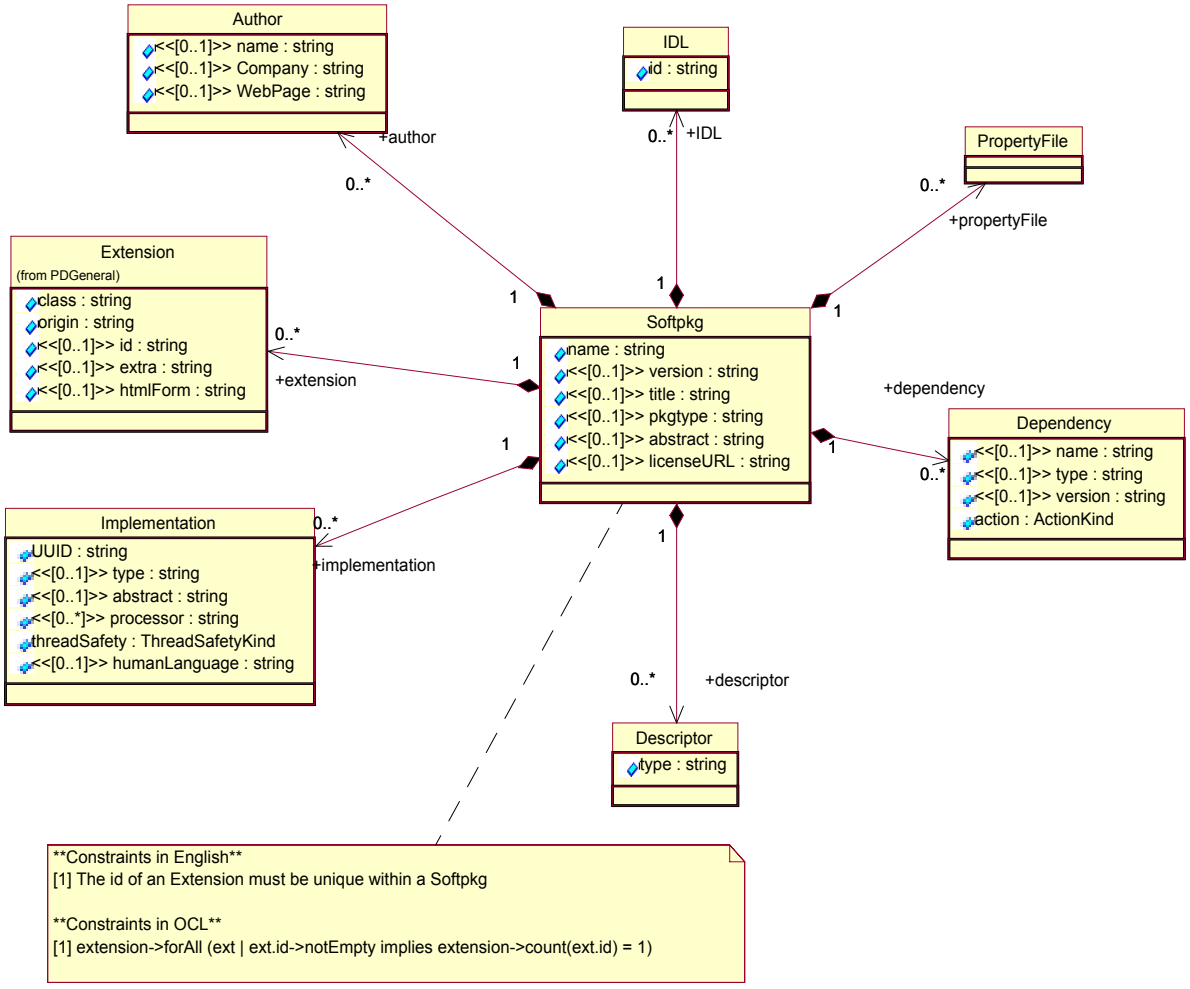

Figure 10-22 Softpkg Top-Level Elements

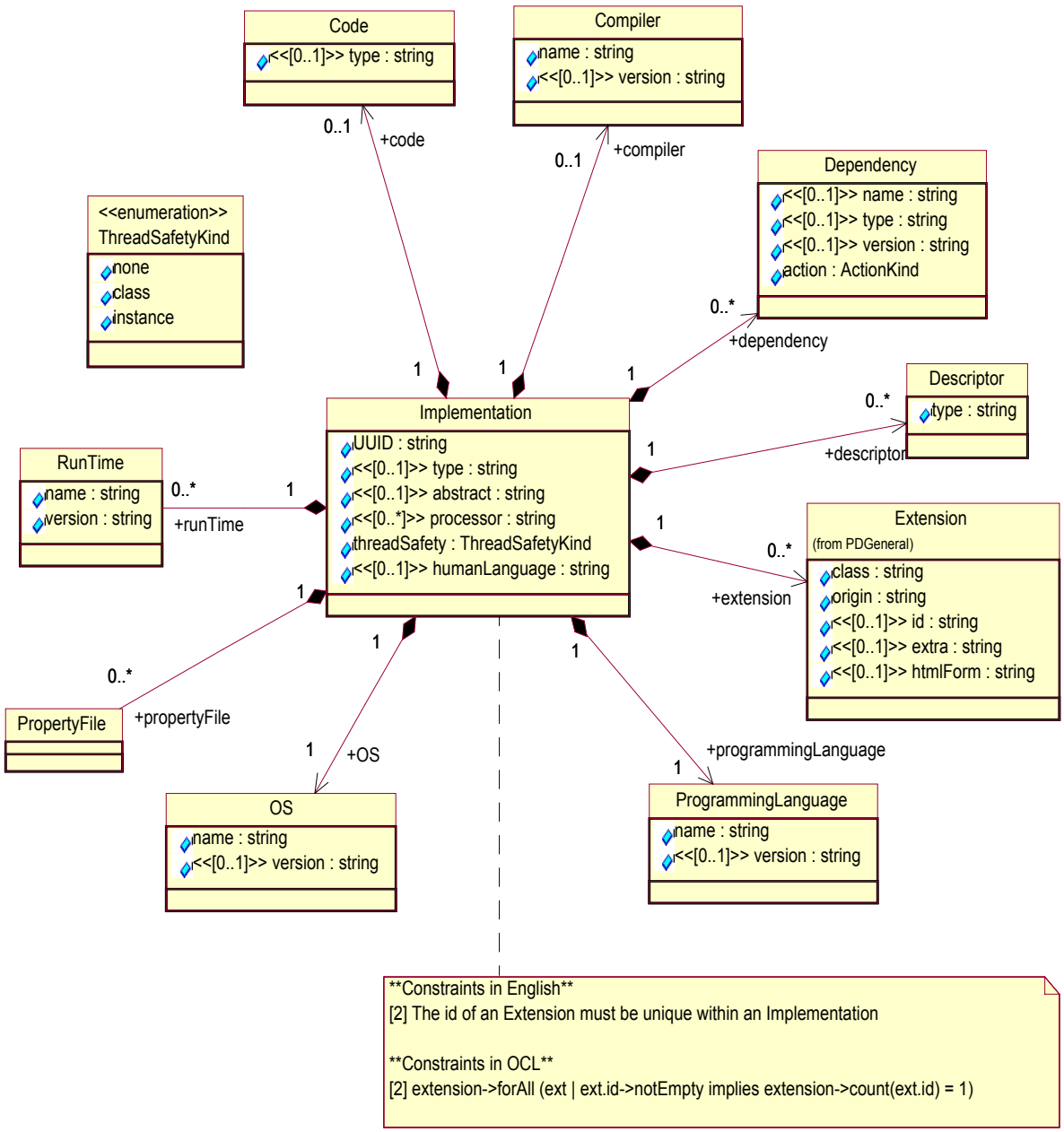

Figure 10-23 The *Implementation* Element

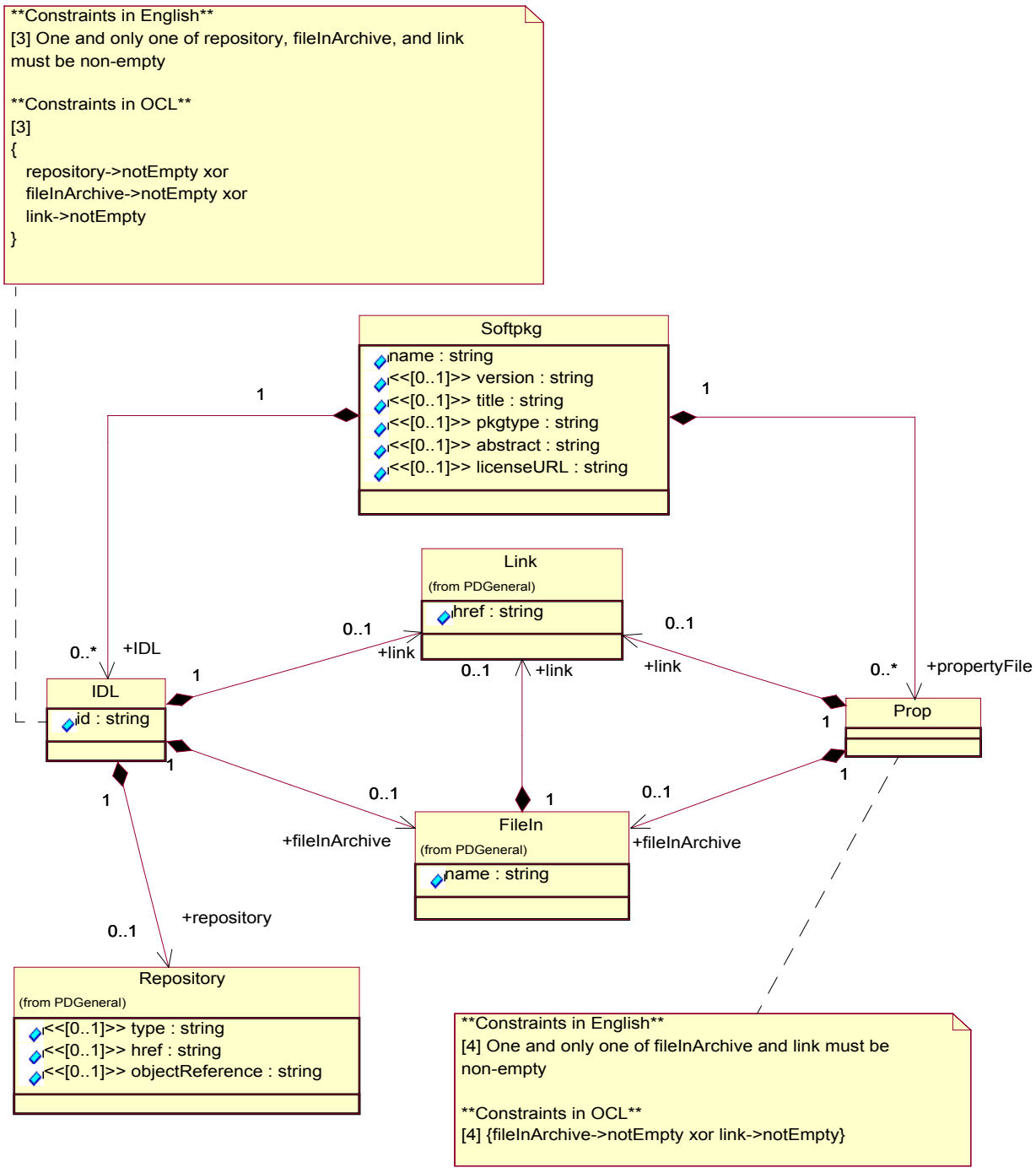

Figure 10-24 The *IDL* and *PropertyFile* Elements

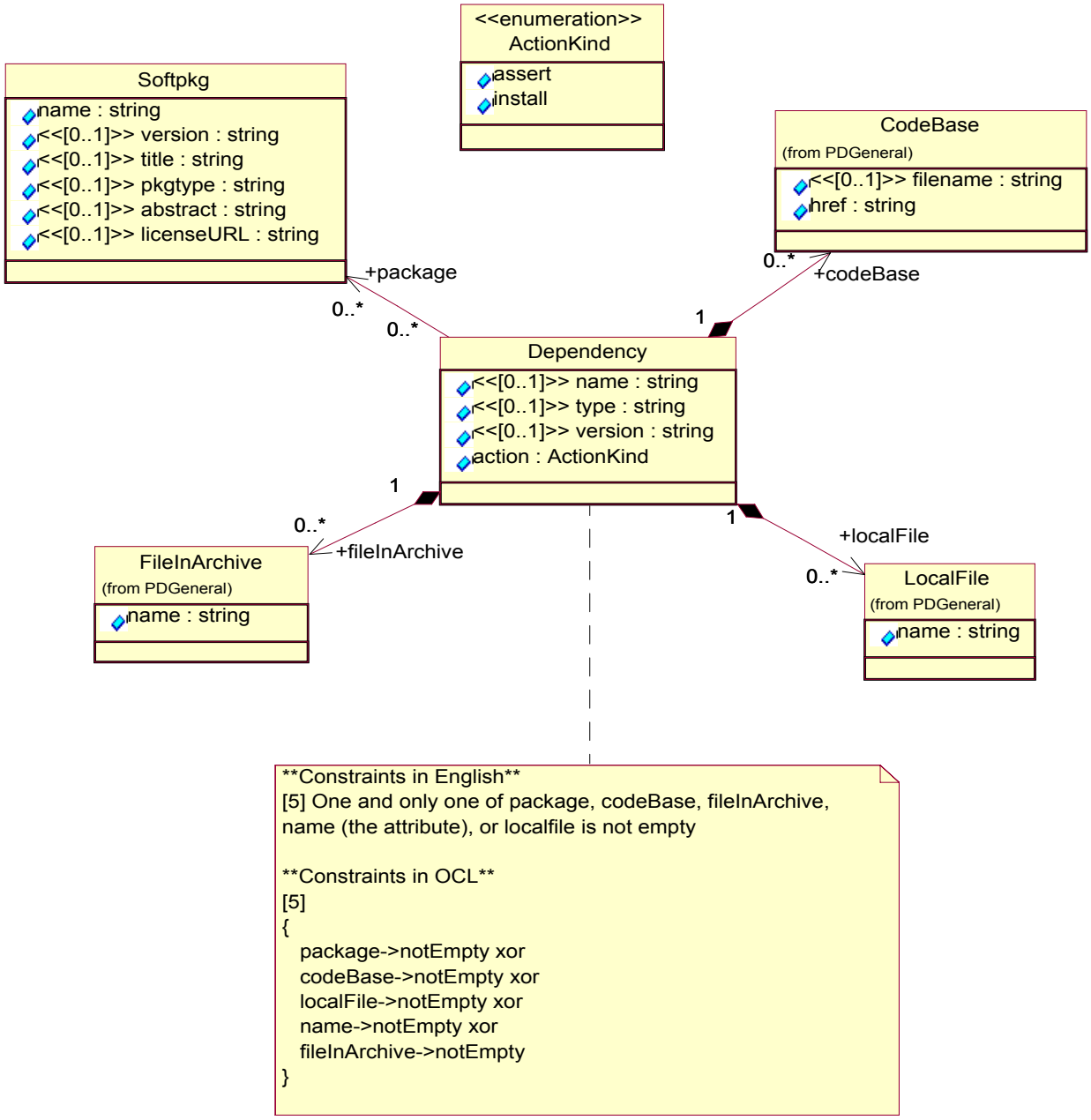

Figure 10-25 The *Dependency* Element

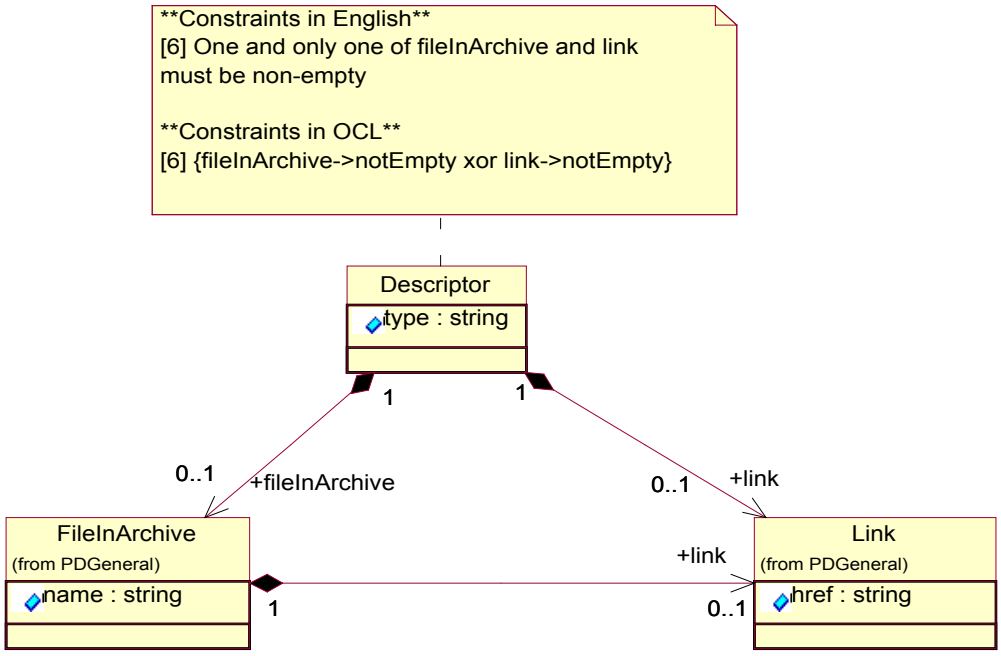

Figure 10-26 The *Descriptor* Element

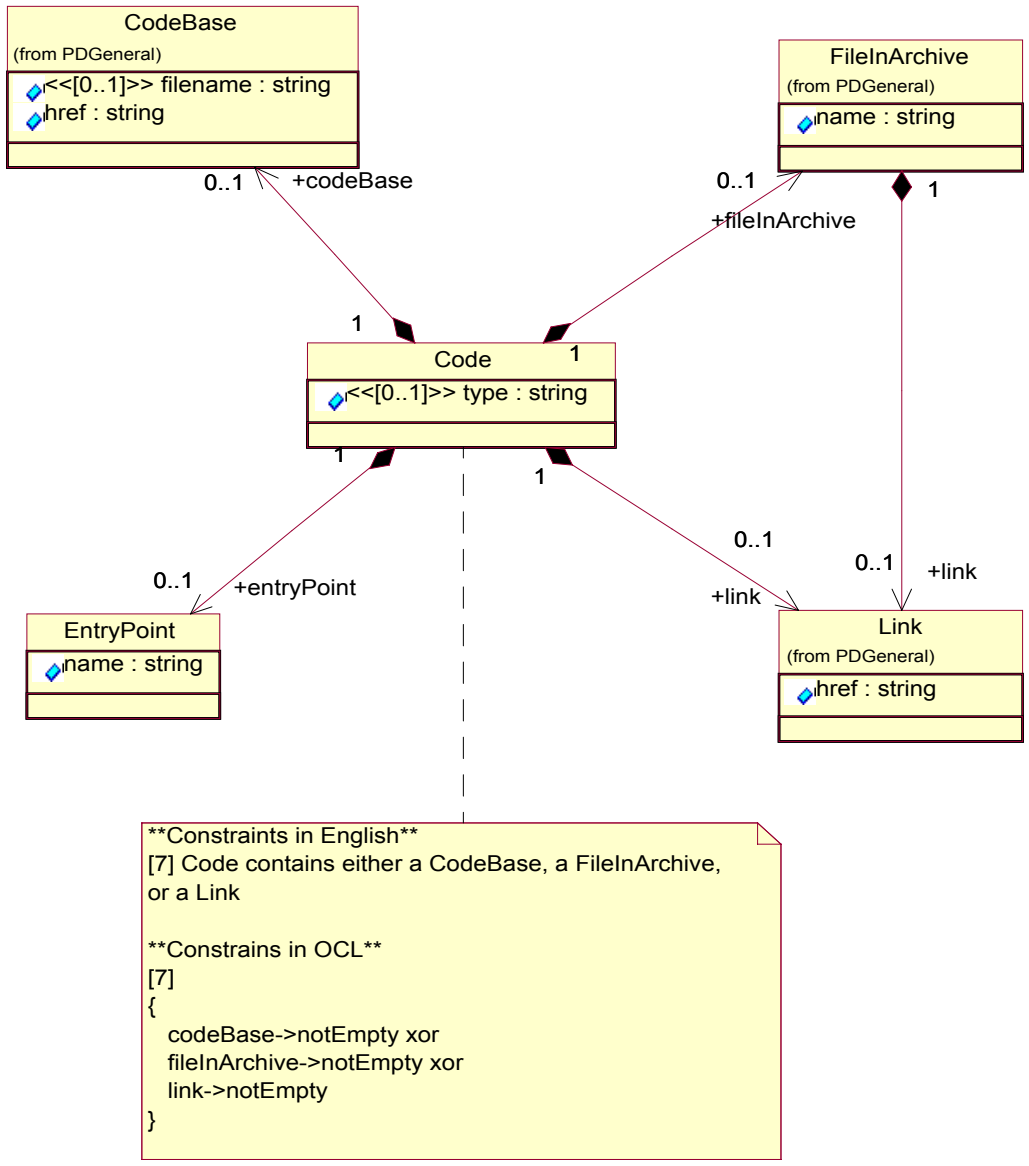

Figure 10-27 The *Code* Element

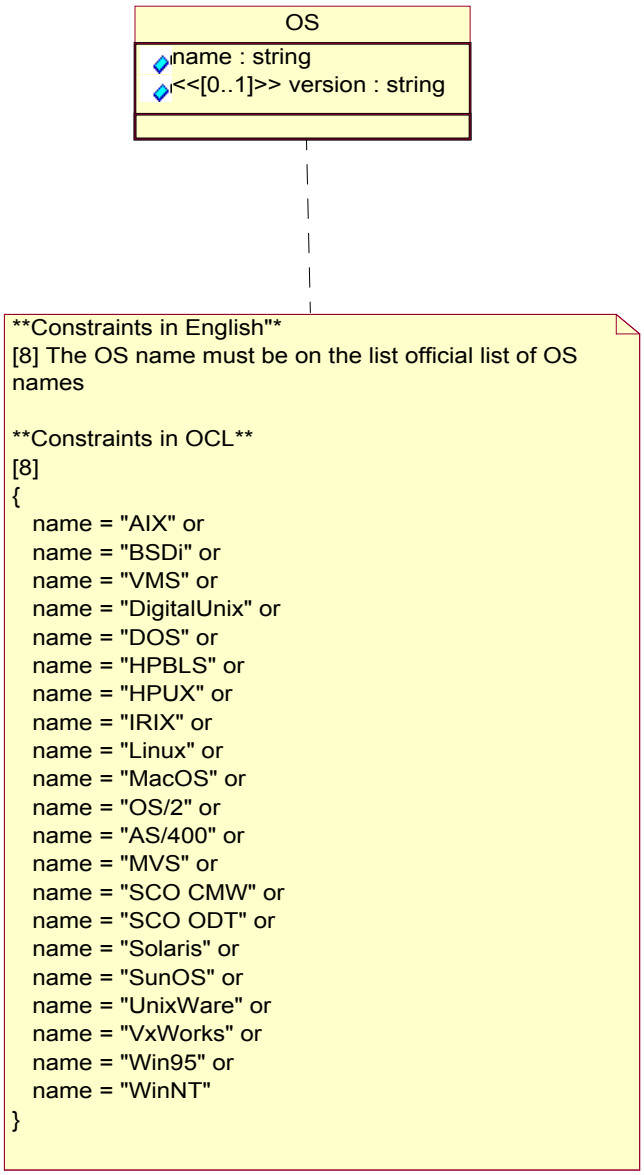

Figure 10-28 The *OS* Element

10.6.3 The Component MOF Package

This Package corresponds to the *Component Descriptor* in Section 9.4 on 243.

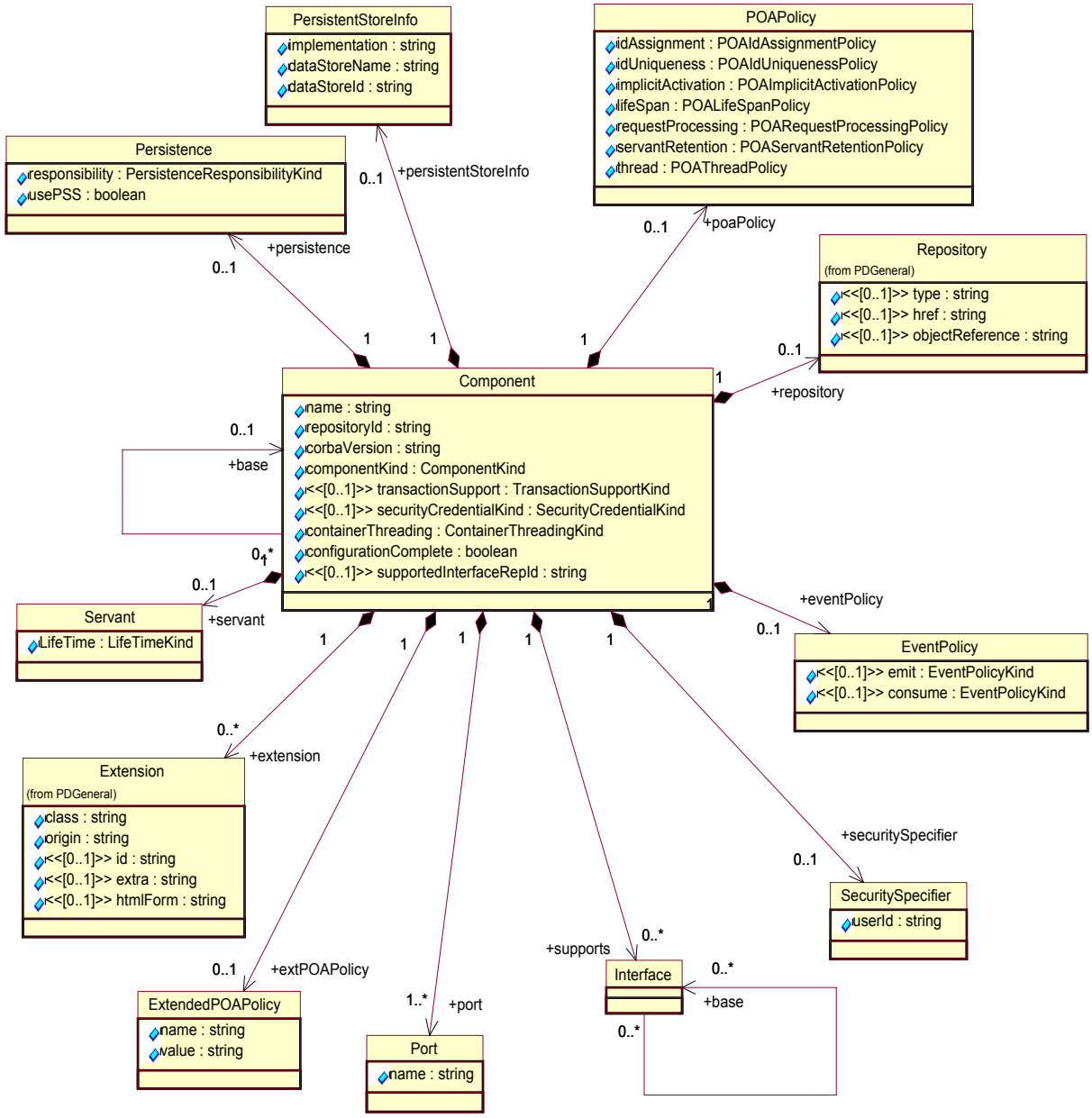

Figure 10-29 Component Top-Level Elements

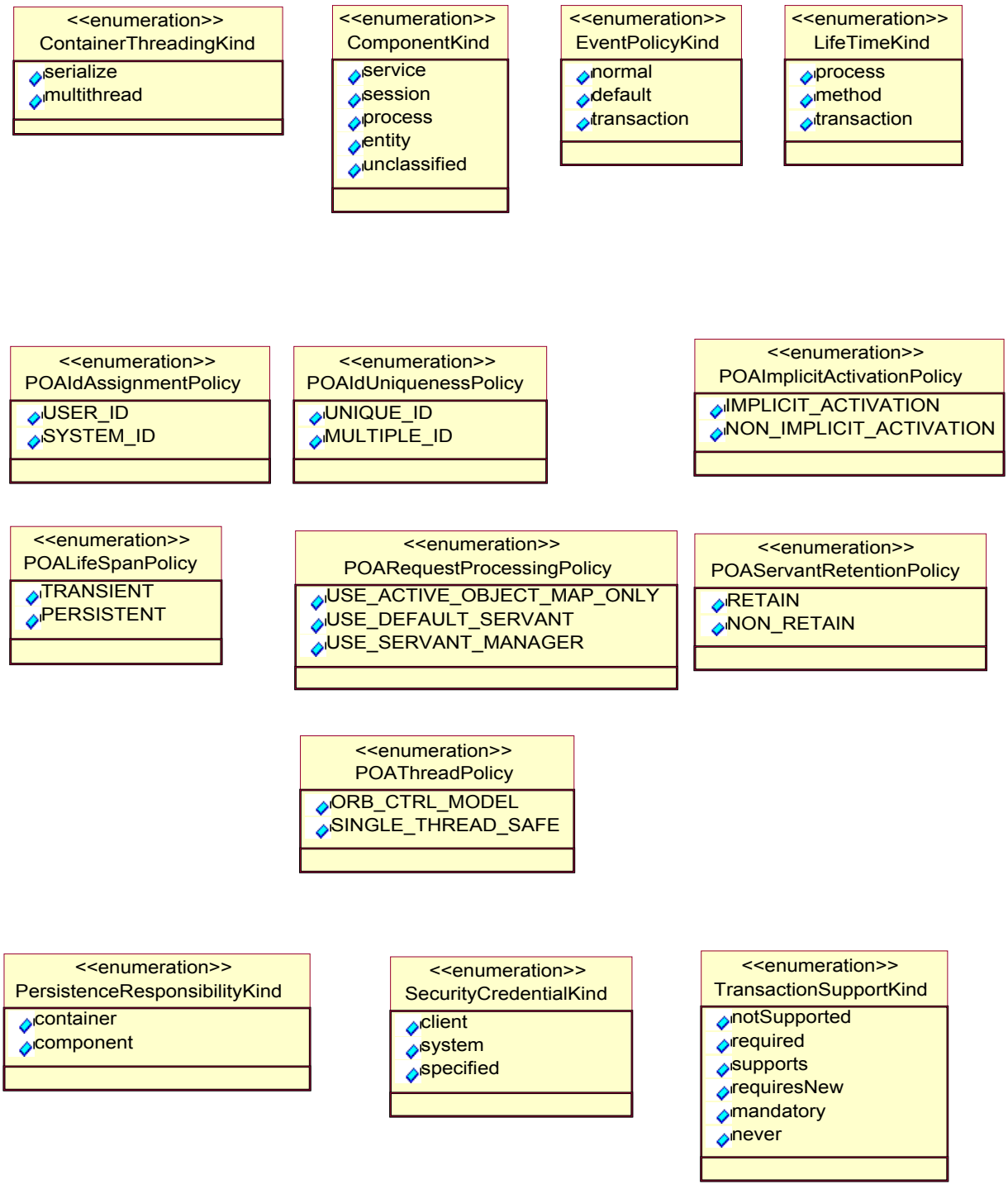

Figure 10-30 Enumerations Used by Top-Level Elements

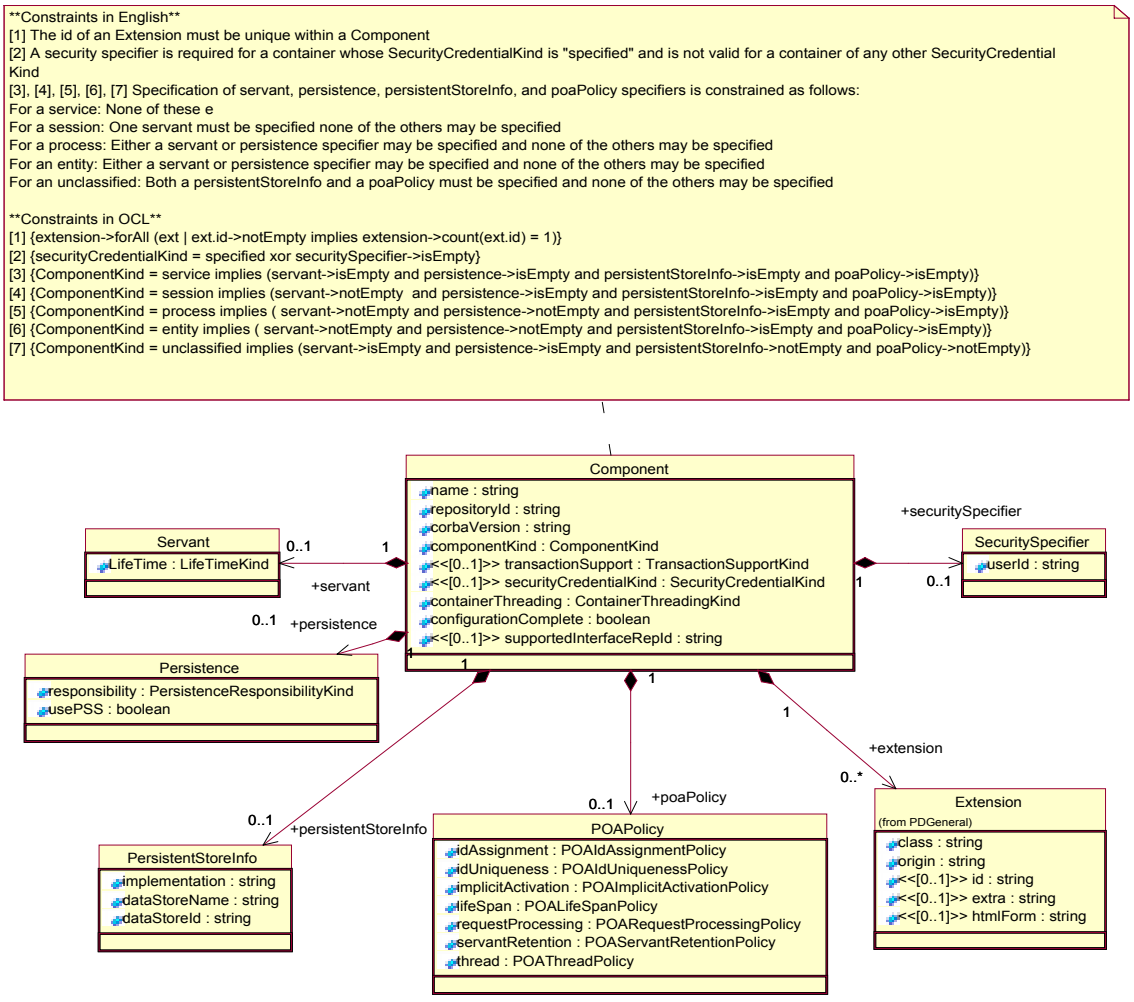

Figure 10-31 Constraints on *Component*

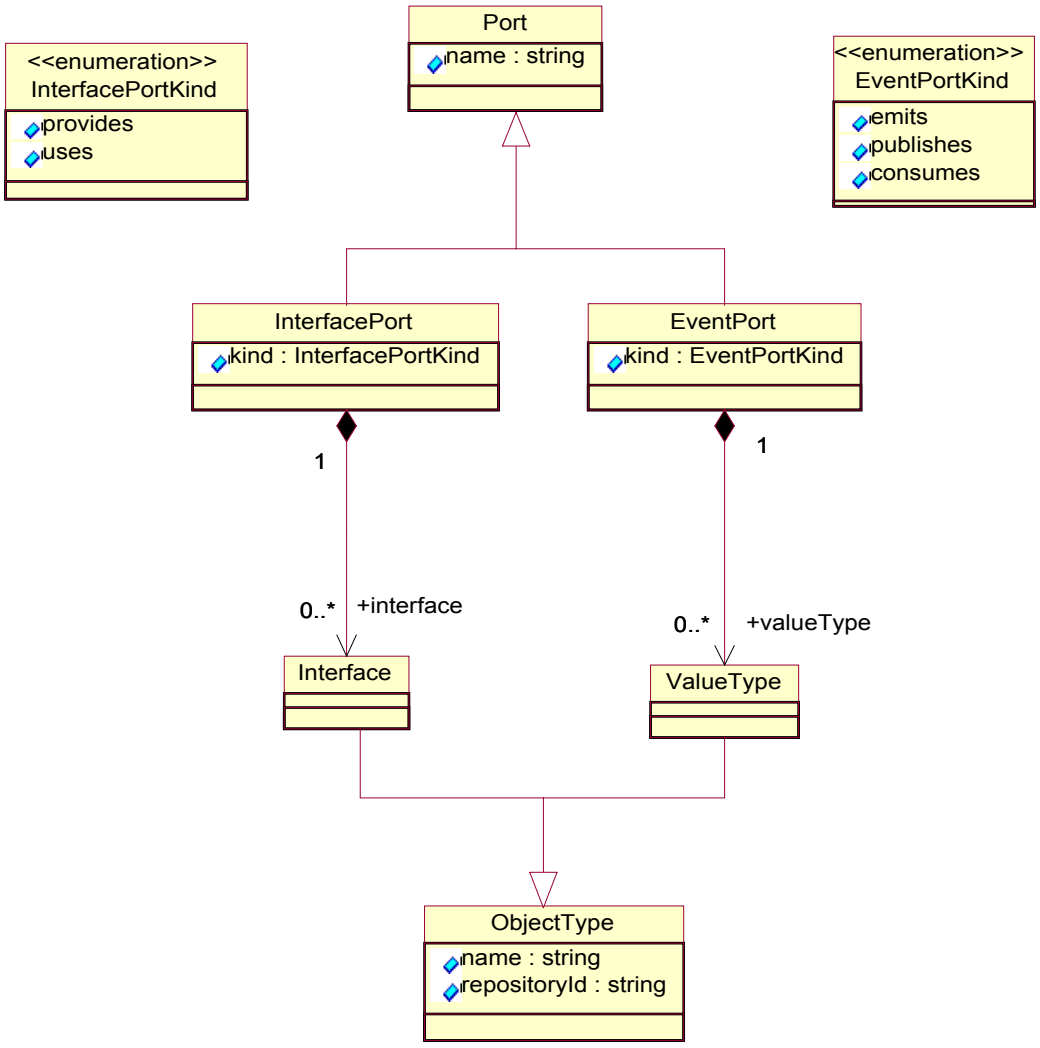

Figure 10-32 Port and Related Elements

10.6.4 The Assembly MOF Package

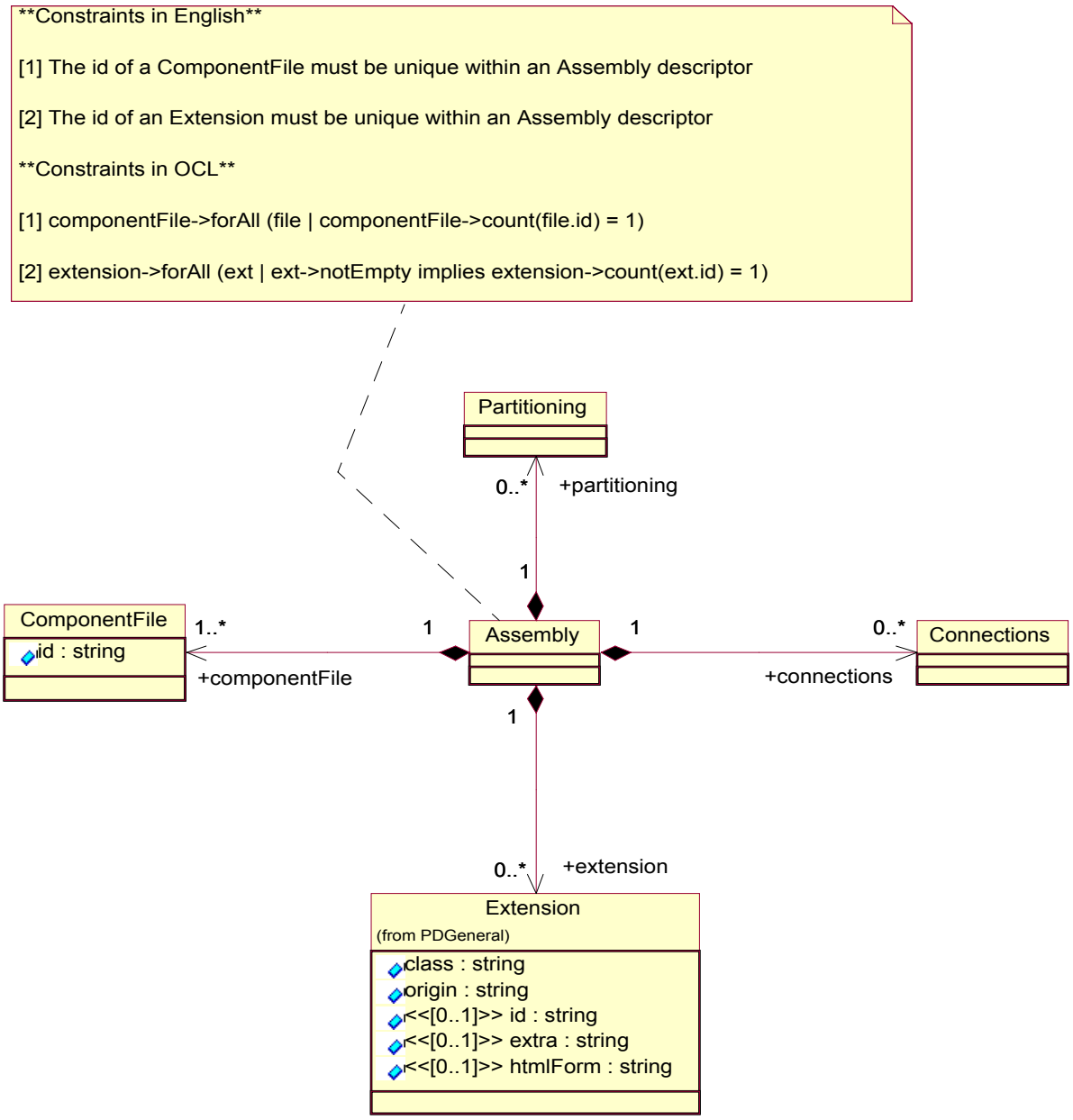

Figure 10-33 Assembly Top Level Elements

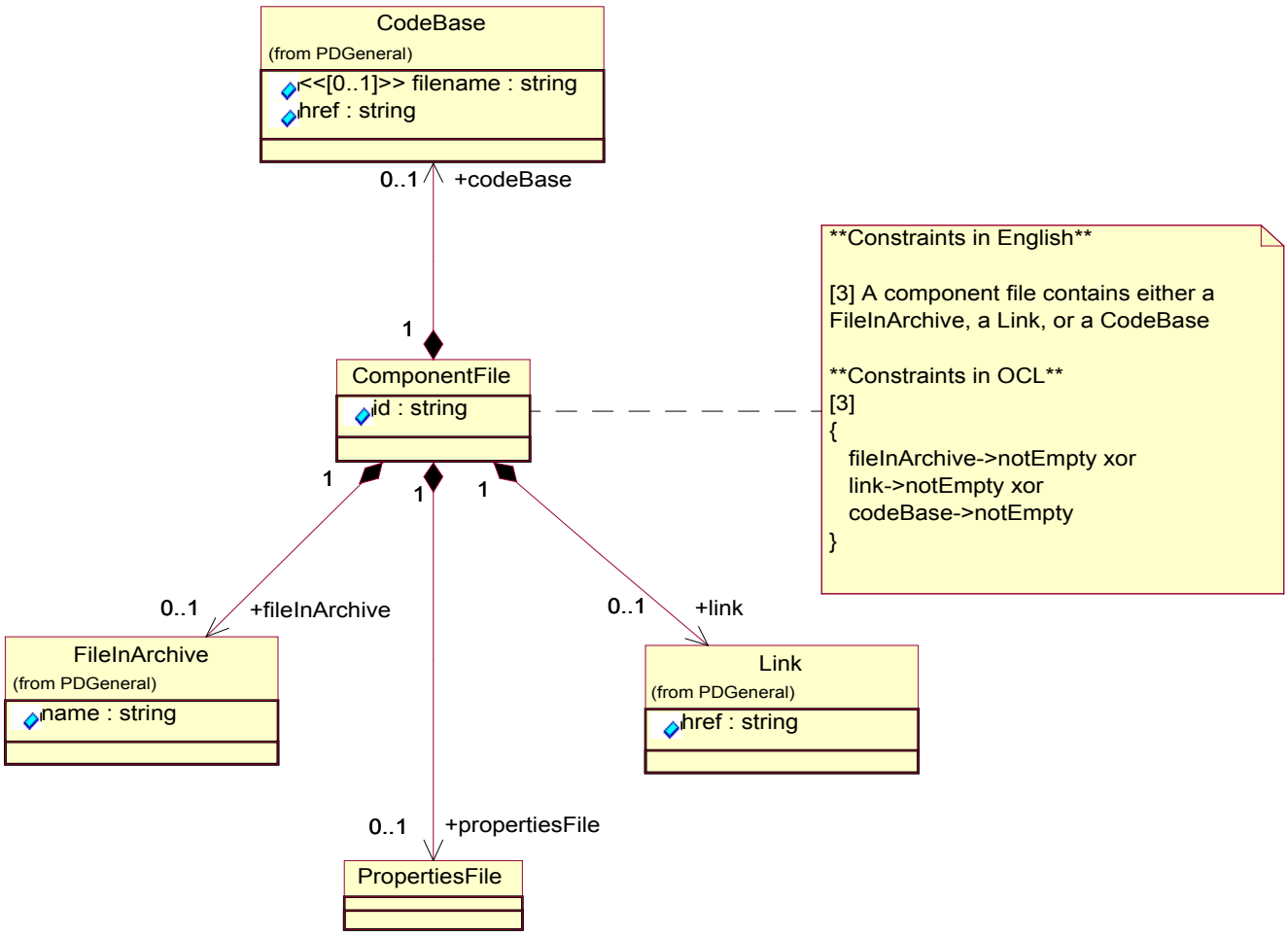

Figure 10-34 The *ComponentFile* Element

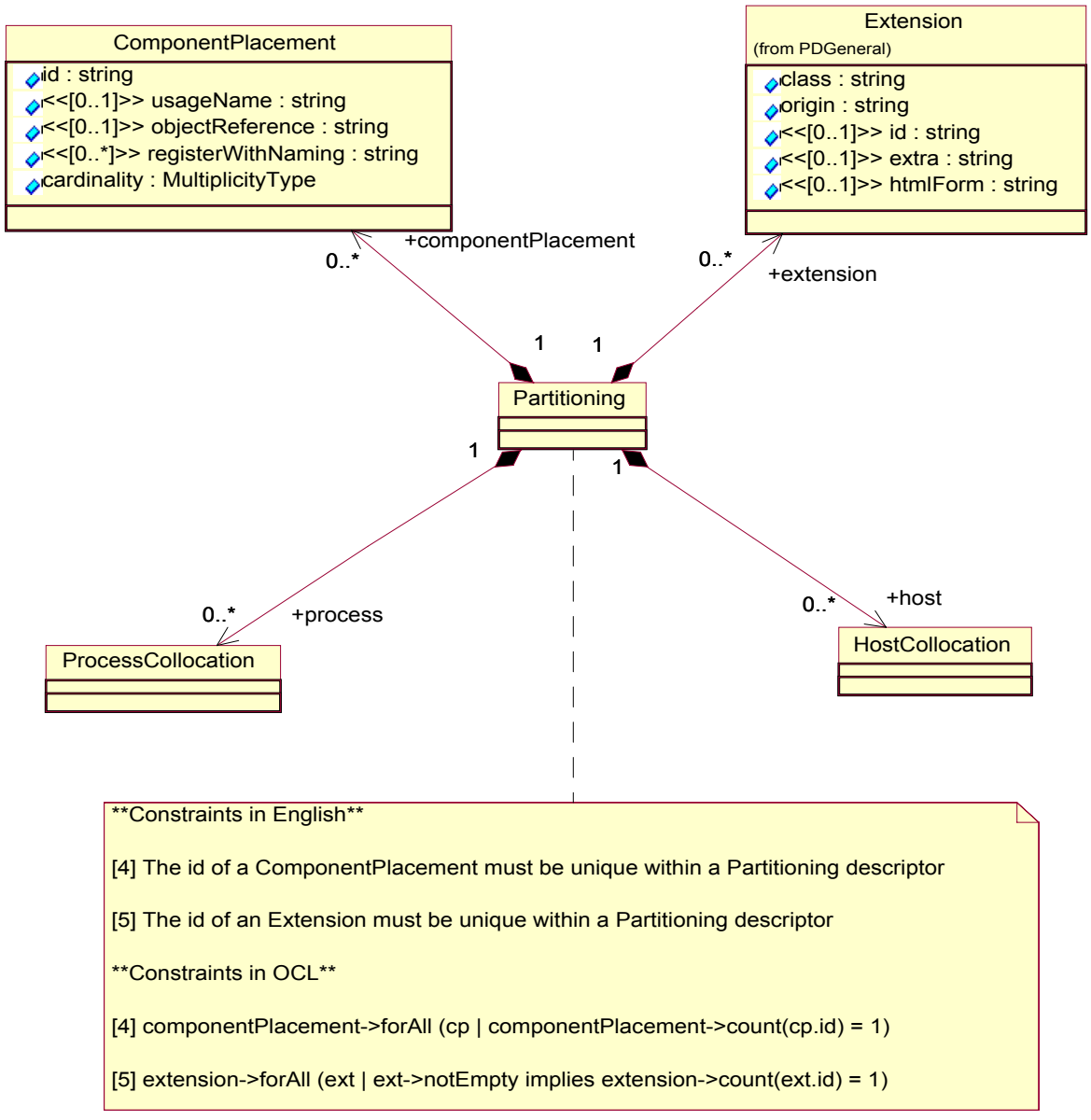

Figure 10-35 The *Partitioning* Element

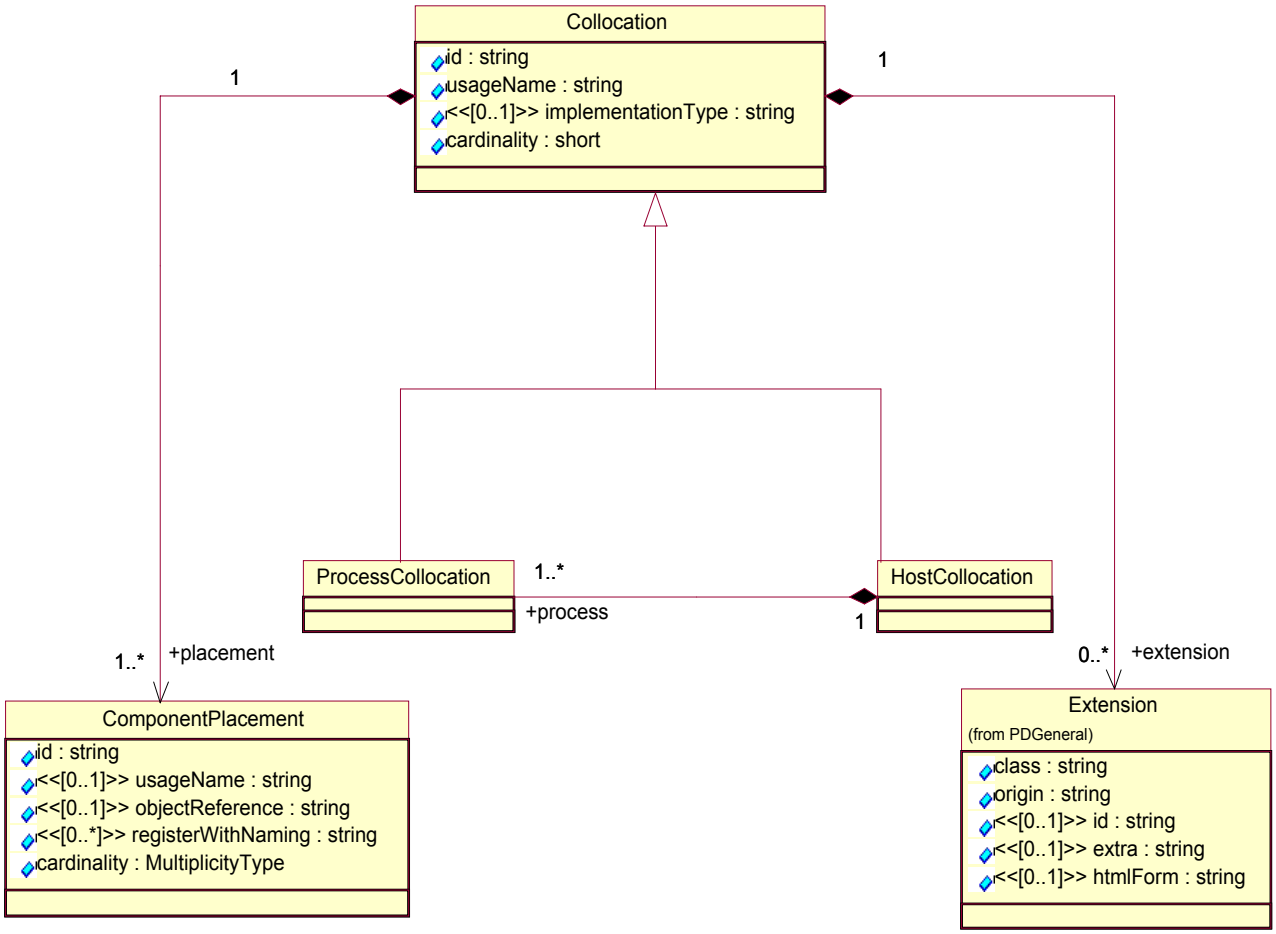

Figure 10-36 Collocation

Note that *MultiplicityType* is a MOF construct that can be used for describing metamodels. It is used in the *ComponentPlacement* element to represent cardinality.

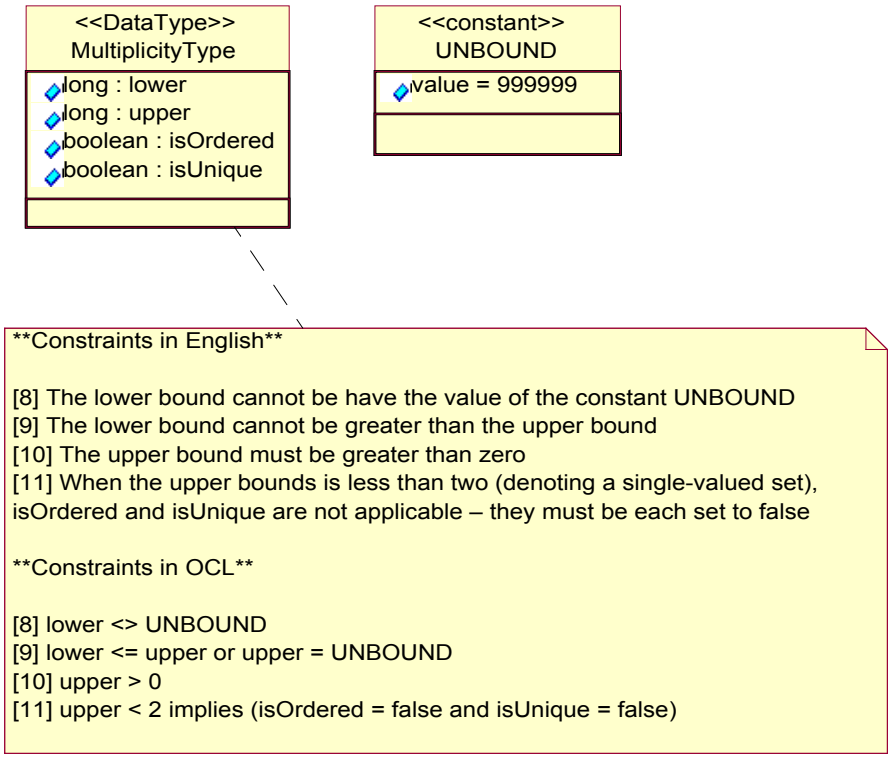

Figure 10-37 MultiplicityType as Defined by the MOF Specification

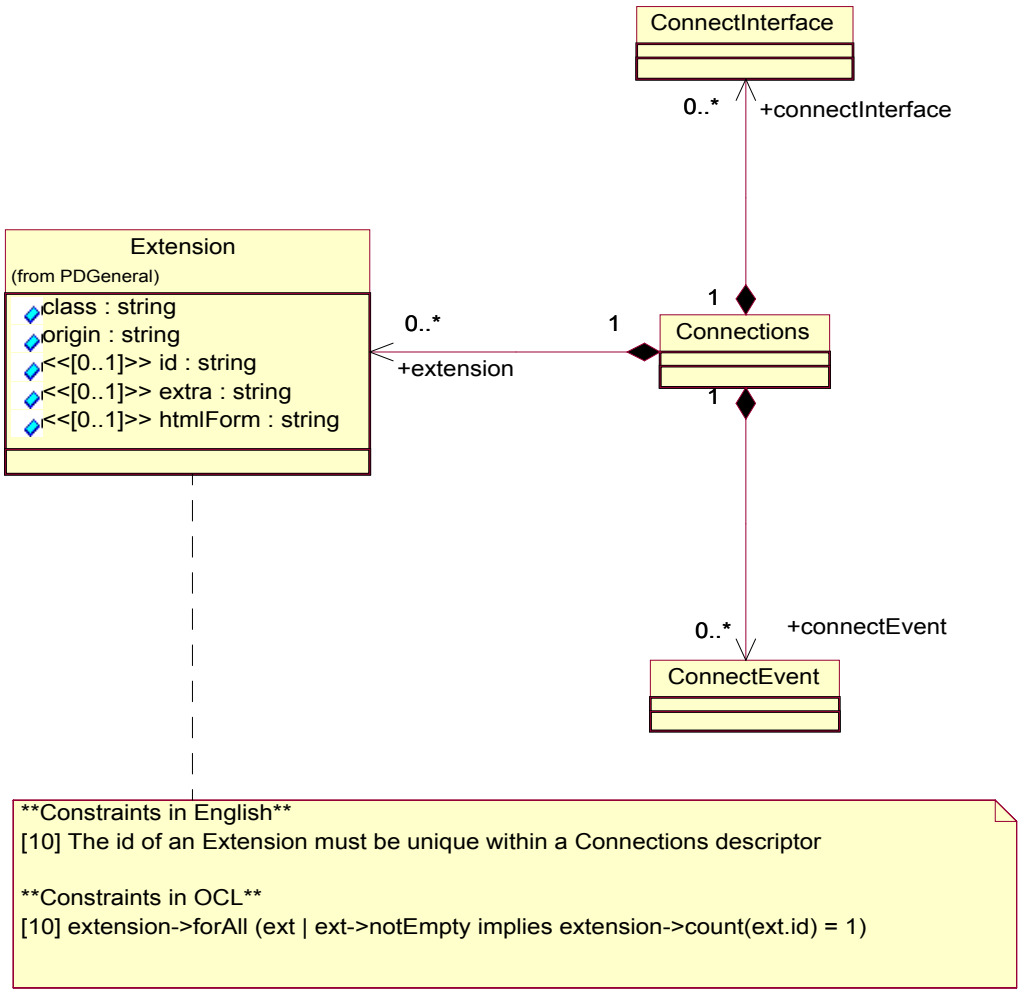

Figure 10-38 The *Connections* Element

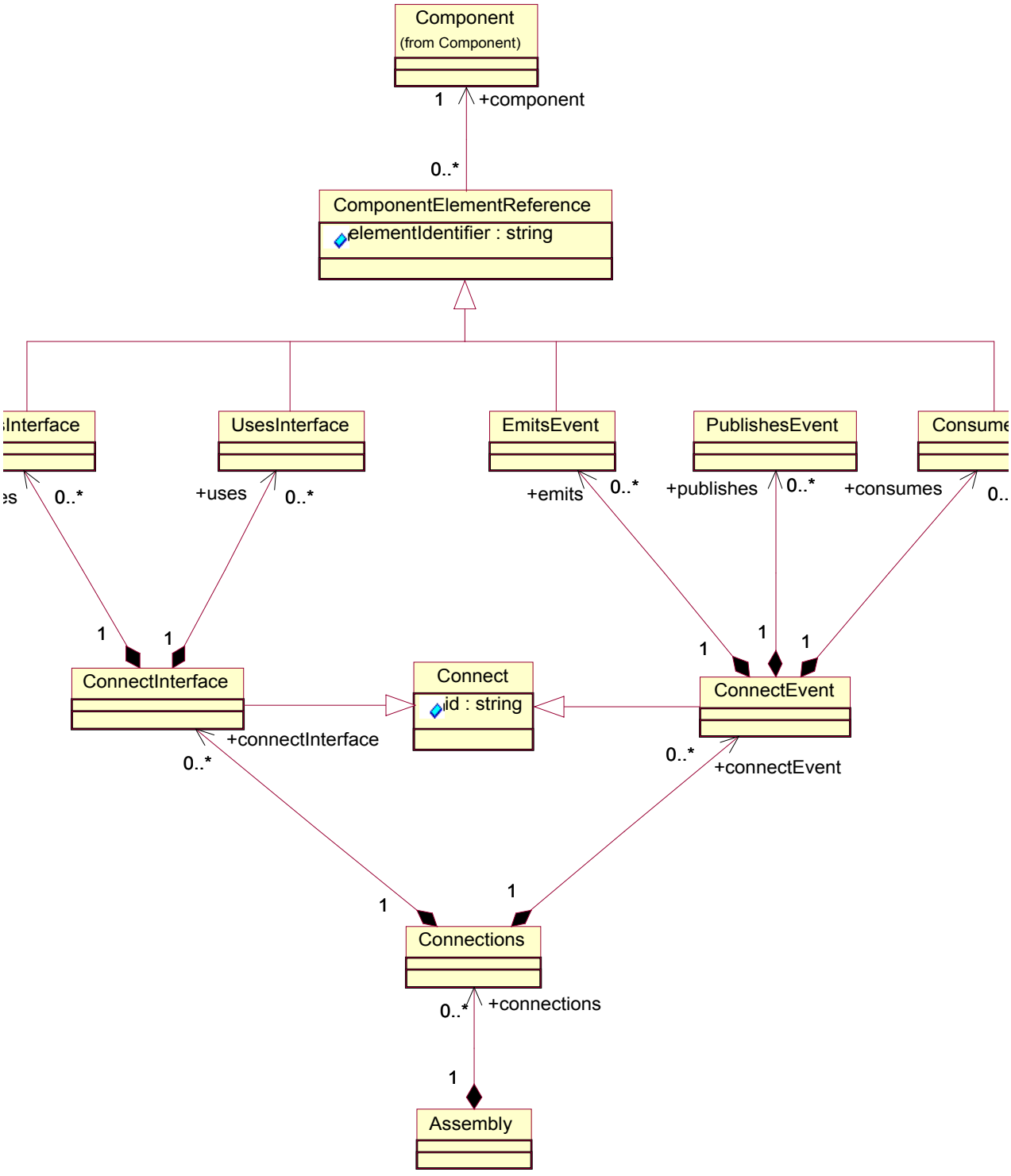

Figure 10-39 Connecting to a Component

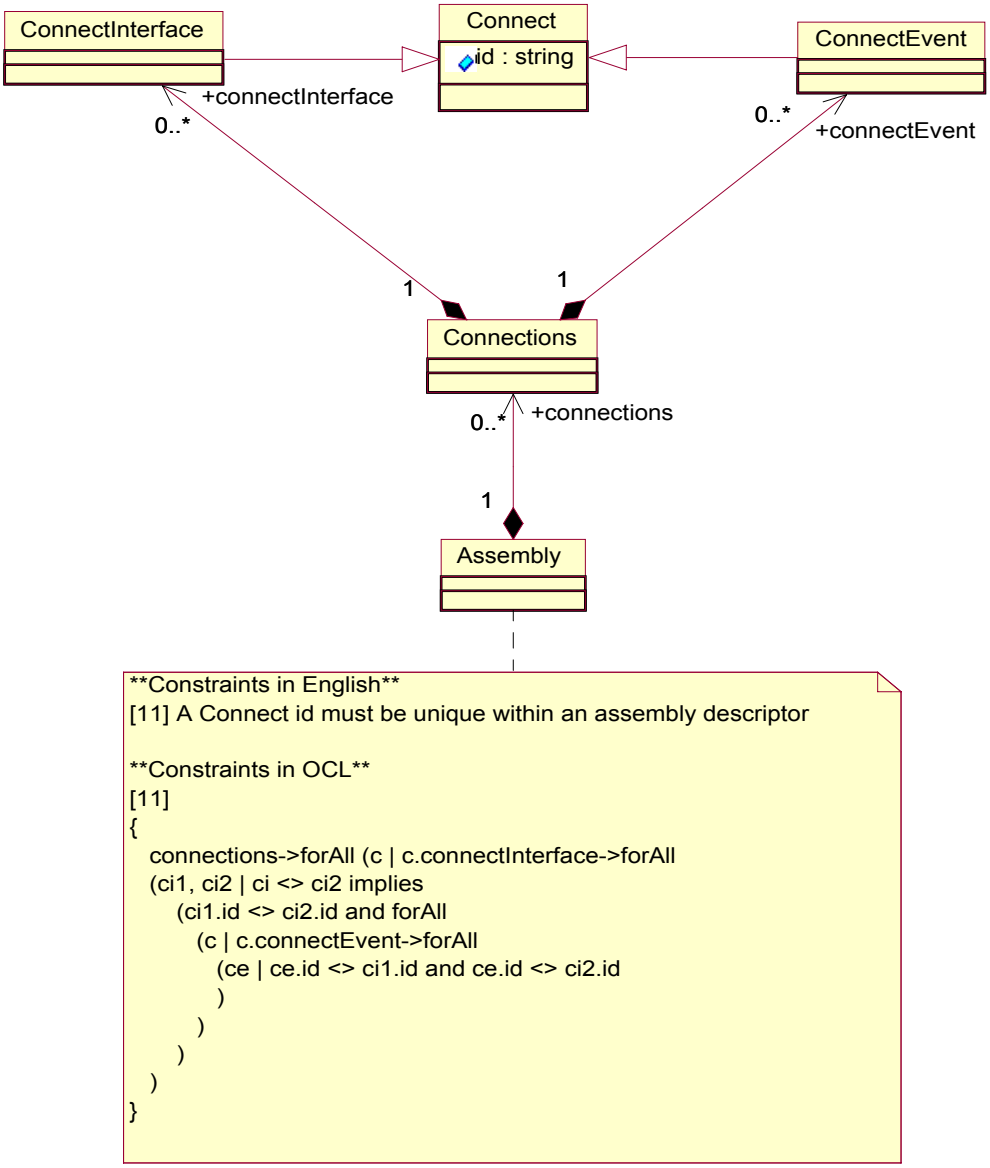

Figure 10-40 Connect id Uniqueness Constraint

10.6.5 The PropertySet MOF Package

This Package corresponds to the *Properties Descriptor* in Section 9.8 on 276. Properties files hold configuration properties for components. The *PropertyFile* element in the *Softpkg* Descriptor and metamodel represents the file itself and does not model the contents. The *Properties Descriptor* describes the contents, and this Package presents a corresponding metamodel. Figure 10-41describes all the elements of this Package.

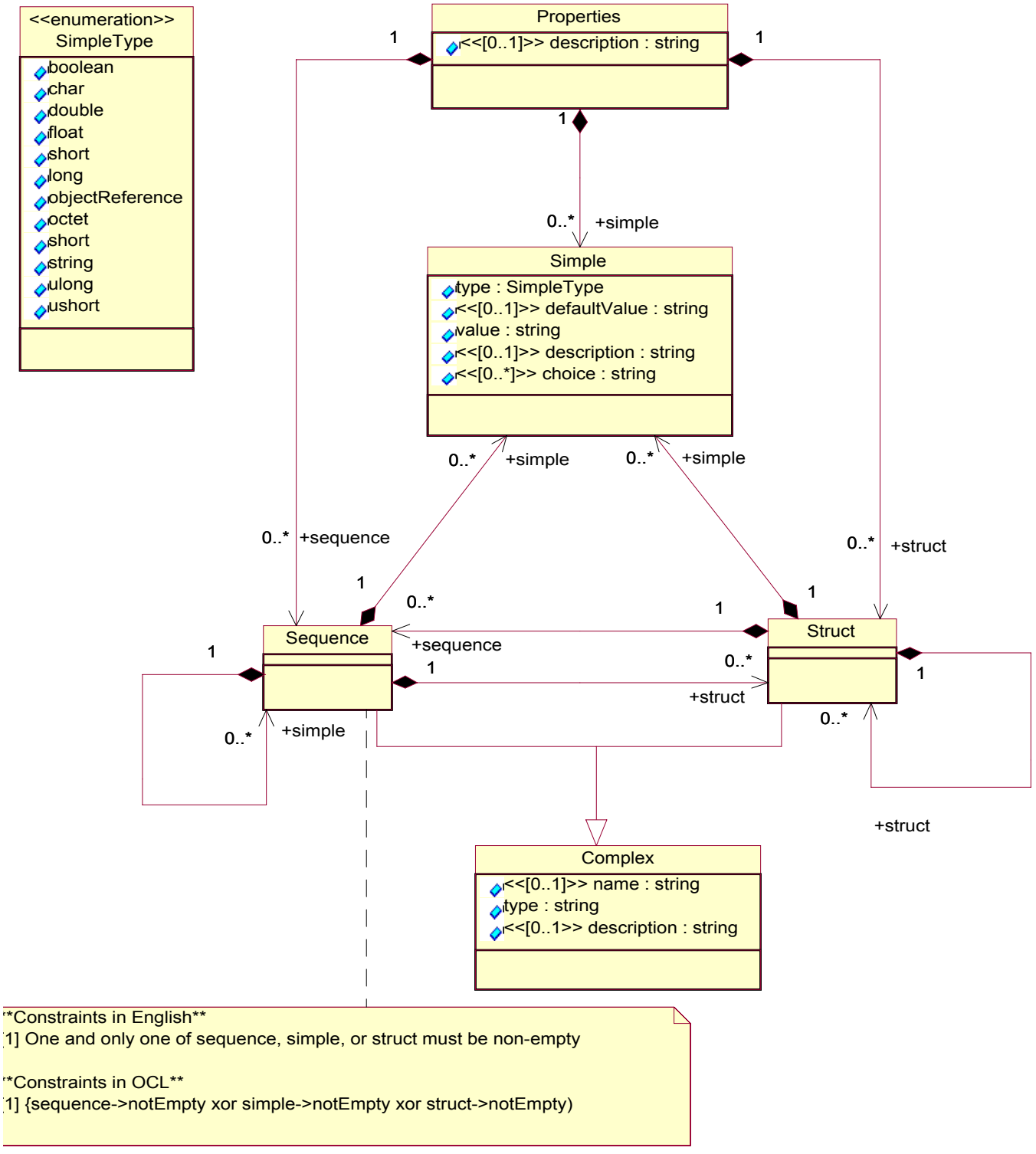

Figure 10-41 The *PropertySet* Package--All Elements

Mapping to Enterprise Java Beans 11

This chapter describes how an Enterprise Java Bean component can be used by CORBA Components. The EJB will have a CORBA Component style remote interface that is described by CORBA IDL (including the Component extensions).

Additionally, it suggests how an Enterprise Java Bean may be deployed in a CORBA Component server. The CORBA Component server (and associated tools) must provide the EJB Environment defined by the Enterprise JavaBeans Version 1.0 Specification in order to host EJBs.

11.1 History of changes

11.1.1 Since 99-02-01

1. EJB Remote and Home and **EJBObject** and **EJBHome** interface mappings updated to the new IDL for Components added in 99-02-05.
2. Example updated to the new IDL and new mappings
3. Change history moved to the front of the chapter to match the other chapters.
4. The section on EJB deployment has been heavily edited and the tables introduced in 99-02-01 have been relocated to this section. The tables relating to the client mapping have been removed as this is now described in the client mapping section.

11.1.2 Since 98-12-02

1. Change history section is introduced.
2. Tables are introduced comparing the EJB contracts with the corresponding CORBA Component contracts.

3. Some advice is given about EJB hosting strategies for CORBA Component Server developers.
4. Terminology changes replaced “Remote” with **EJBObject**.
5. Font changes now have programming terms in 10 pt Arial bold font (IDL format).
6. Removed the discussion of local dispatch. RMI/IIOP is required. It is up to the vendors to implement it in an effective way when the dispatch is local.
7. Added information on deploying EJBs in a CORBA Component Server. Discussion of Session Bean deployment is TBD.
8. Editorial changes made in anticipation of the EJB 1.1 specification - principally the removal of issues that are being addressed in this specification.

11.2 Enterprise Java Beans Compatibility Objectives and Requirements

The most important objective is to allow the creation of distributed applications which mix CORBA Components running in CORBA Component servers with EJBs running in EJB servers. These components are bound together by synchronous and asynchronous method invocations that are mediated by ORBs. This objective allows a developer to create an application by reusing existing Components of either kind. This requires that CORBA Components have EJB facades, and that EJB components have CORBA Component facades for their remotely accessible behaviors. It also requires that value objects of one kind (e.g. Keys for EJB) have counterpart value objects of the other kind. It also requires that CORBA Components accessible via **CosNaming** have their EJB facades accessible via **JNDI**, and vice versa.

The next most important objective is to allow a CORBA Component server to serve both CORBA Components and EJBs. This allows applications which mix CORBA Components and EJBs to reside in the same server for performance, security, and other reasons. This adds requirements to the server to manage both kinds of components and to provide local synchronous and asynchronous method calls.

The third most important objective is to allow CORBA Components written in Java and following the EJB patterns to be deployable in EJB servers. This allows a developer to create a component knowing that it is usable either as an EJB or as a CORBA Component. The practical consequence of this objective is that the CORBA Component is, in fact, an EJB. There are two possibilities:

- The component is just an EJB and no more;
- The component has an EJB personality and a CORBA Component personality and these are not the same. In the most general case, the component behaviors can be assigned to one of three sets: {EJB - CORBA Component}, {CORBA Component - EJB}, {CORBA Component and EJB}. The component must be capable of detecting its operating environment and enabling or disabling behaviors accordingly.

11.3 EJB Facades for EJBs

This facade allows an EJB deployed in a CORBA Component server to appear as an EJB Remote interface in another EJB (which may or may not be deployed in a CORBA Component server).

The JDK 1.2 from Sun provides an implementation of RMI/IIOP based on the Java Language to IDL Mapping (ptc/98-10-02, subsequently called RMI/IIOP). This implementation can be used by a Java client wishing to access EJBs hosted by a CORBA Component server.

An EJB running in an EJB server can access an EJB running in a CORBA Component server provided that the EJB server contains a CORBA ORB supporting the RMI/IIOP specification.

An EJB running in a CORBA Component server can access another EJB running in a CORBA Component server by way of the EJB remote interface implementation that is defined by the RMI/IIOP.

The container tools are expected to provide an implementation of the Handle class for an EJB. The Handle is expected to be able to locate an EJB unless it has been destroyed or is a Session Bean which has not survived a crash or server imposed lifetime. Thus Handle behavior may differ from one EJB server to the next.

11.4 CORBA Component facades for EJBs

This facade is based on the Java Language to IDL Mapping. However, it includes the Component extensions to IDL. There are two ways that this facade may be created:

1. A new RMI Compiler tool can be created which directly generates the Component style proxies by introspection of the Java classes and interfaces;
2. An existing RMI Compiler tool can be used to generate non-Component IDL which is then processed by a second tool to generate Component level IDL. The second tool may directly generate the Component style proxies or it may generate wrappers around non-Component proxies.

The specification does not restrict the implementors freedom to use either of these patterns or any other functionally equivalent pattern.

11.4.1 Java Language to IDL Mapping

An EJB has a natural mapping to CORBA IDL by way of the Java Language to IDL Mapping Specification. The reader is assumed to be familiar with this specification, whose major aspects are repeated here for convenience. The Home and Remote interfaces can be mapped to CORBA IDL without significant restrictions.

- EJB requires the Remote and Home interfaces to inherit **java.rmi.Remote**. EJB requires all methods on the Remote and Home interfaces to throw **java.rmi.RemoteException**. These are requirements of the Java Language to IDL mapping also.

- get- and set- name pattern names are translated to IDL attributes.
- IDL generated methods have only **in** parameters (but these can include object references to remote objects, allowing reference semantics normally obtained by using parameters of type **java.rmi.Remote**).
- Java objects that inherit from **java.io.Serializable** or **java.io.Externalizable** are mapped to a CORBA **valuetype**. All object types appearing in RMI remotable interfaces must inherit from these interfaces or from **java.rmi.Remote**. EJB **Key** and **Handle** types must inherit from **java.io.Serializable**.
  - However, the mapping does NOT require that methods on such objects or constructors be mapped to corresponding IDL operations on **valuetypes** and **init** specifications. The developer is expected to select those methods which should be mapped to IDL operations, and the method signatures must meet the requirements of the mapping.
  - Objects which inherit from **java.io.Externalizable** or which implement **writeObject** are understood to perform custom marshalling and the corresponding custom marshallers must be created for the CORBA **valuetype**.
- Arrays are mapped to "boxed" CORBA **valuetypes** containing sequences because Java arrays are dynamic.
- Java exceptions are subclassable; IDL exceptions are not. Consequently a name pattern is used to map to IDL exceptions. The Java exception object is mapped to a CORBA **valuetype**. The CORBA **valuetype** has an inheritance hierarchy like that of the corresponding Java exception object.
- Some additional programming is required to define Java classes (including EJB implementations) that are accessible via RMI/IIOP. This is to account for the fact that IIOP does not support distributed garbage collection. However, note that these operations would normally be provided in the **EJBObject** implementation supplied by the EJB server.

11.4.2 EJB to CORBA Component IDL mapping

The rules for mapping the public interfaces to a CORBA Component declaration are also straightforward. In general, the CORBA Component will support an interface that is the RMI/IIOP map of the Remote interface of the EJB. However, this interface is named ***XXX*Default**, where ***XXX*** is the name of the EJB Remote interface. In the following text, this generated interface will be referred to as the *Default* interface of the Component.

11.4.2.1 Operations on EJBObject

These operations must be mapped to the equivalent operations on ComponentBase and on the generated Component IDL *default* interface as follows:

- **getEJBHome** is mapped to **get_home** on **ComponentBase**.
- The **getHandle** operation on an EJB object reference is not available on the corresponding CORBA Component object reference.

EJB does not support persistent object references. The **Handle** object provides a storage form (via serialization) for the object reference it encapsulates. It also provides a **getEJBObject** operation to recreate the object reference from the reserialized **Handle**. These functions correspond closely to the CORBA **string_to_object** and **object_to_string** operations. There are differences in the behavior of Handles compared with the behavior of CORBA Object References. These details are discussed later in this chapter.

- The **getPrimaryKey** operation is mapped to a **get_primary_key** operation on the *default* interface supported by the CORBA Component, provided that the EJB Home declares the **findByPrimaryKey** operation.
- The **isIdentical** operation is mapped to an RMI/IIOP equivalent operation on the *default* interface supported by the CORBA Component.
- The **remove** operation is mapped to the **destroy** operation on **ComponentBase**.

11.4.2.2 Operations on EJBHome

- **getEJBMetaData** is mapped to **get_component_def** on **HomeBase**

EJBMetadata provides references to the Class objects for the Primary Key, Home and Remote interfaces of the EJB. The Class object supports reflection, allowing the user to determine the signature of the interfaces at runtime. The **ComponentDef** interface obtained from the Interface Repository provides the equivalent function for the CORBA Component programmer.

The Class object also supports dynamic dispatch (on the Remote interface proxy). The equivalent CORBA function would be to use DII on the Component/Home Object reference. CORBA does not support dynamic dispatch of operations on valuetypes, so there is no mechanism that would correspond to a Java dynamic dispatch of an operation on a Primary Key class.

EJBMetadata also allows navigation to the **Home** from the Metadata via the **getEJBHome** operation. This is unfortunate. If multiple Homes of the same type are deployed, each Home must have its own copy of the Metadata in spite of the fact that all copies are identical. This specification does not allow navigation from the **ComponentDef** to the **HomeBase**.

- **remove (Handle handle)** is not mapped.
- **remove (Object key)** is mapped to **destroy_by_*KKK* (in *KKK* key)** on the Component Home interface, provided that the EJB Home declares the **findByPrimaryKey** operation. The **remove (Object key)** operation is also mapped to the **destroy_component (in ComponentBase comp)** operation on **HomeBase**.

11.4.2.3 Operations on the Remote Interface

- The EJB Remote interface declaration is used to create a **supports** declaration and the corresponding IDL for the primary interface of the Component. The identifier of this supported interface on the component is ***XXX*Default**, where ***XXX*** is the name of the Remote interface. The form of the declaration is **component *XXX* supports *XXX*Default**.
- Each operation on the Remote interface is mapped under RMI/IIOP to an equivalent operation on the ***XXX*Default** interface. Note that pairs of **get*XXX*** and **set*XXX*** methods in the Remote interface will be mapped to IDL attributes. Because the IDL introduced by the Component proposal permits attributes to raise exceptions, it is now possible to map such method pairs even though they may raise exceptions.

11.4.2.4 Operations on the Home Interface

- The EJB Home operations prefixed "**create**" are mapped into **Home** factory declarations in IDL. The full name of the operation, e.g. **create*XXX***, becomes the factory operation identifier in the **<factory_dcl>**. The Java parameters of the operation are mapped to their corresponding IDL types and names as defined by RMI/IIOP.
- An EJB Primary Key class is mapped to a CORBA **valuetype** using the mapping rules in RMI/IIOP. This **valuetype** will be declared in the IDL for the Home as the primary key **valuetype** for the Component. The key **valuetype** will inherit from **Components::PrimaryKeyBase**.
- The Home operation named "**findByPrimaryKey**" is mapped into the **find_by_*key_name*** operation on the Component Home interface, where **key_name** is the name of the **valuetype** class that the EJB Primary Key is mapped into.
- The other Home operations prefixed "**find**" are mapped into **ComponentHome** finder operations in IDL. The EJB operation name, e.g. "**findXXX**" becomes the identifier of the corresponding finder declaration and the parameter types and names are mapped in accordance with RMI/IIOP.
- Finder and Creator EJB operations which return an RMI style object reference are mapped into Component IDL operations which return a CORBA Component Object Reference.
- Finder EJB operations which return a Java Enumeration are mapped into CORBA Component operations which return an IDL Object Reference to an interface of type **Enumeration**. This interface is declared as:

```
interface Enumeration {
    boolean has_more_elements();
    ComponentBase next_element();
};
```

The Enumeration interface is just the RMI/IIOP image of the Java Enumeration class as defined in the JDK 1.1.6+. Sun has said that they intend to replace this with the JDK 1.2 (Java 2.0) Collections in a future version of

the EJB specification. Subsequent to such a specification being issued, the CORBA Components specification will be updated to correspond.

- In the EJB 1.0 specification, no operations other than creators and finders are allowed on Homes. We anticipate that this restriction will be relaxed or removed in some future EJB specification. Consequently all other Home operations are mapped into **<export>** declarations in IDL for the Home declaration, as specified by RMI/IIOP.
- The EJB Remote Interface name becomes the identifier of the **<component_header>**.
- The EJB Home Interface name becomes the identifier of the **<home_header>**.

11.4.2.5 Other mapping rules

The EJB specification rules out the use of Java Bean style event programming, and hence event listeners on EJBs. Programmers may implement a similar facility by directly programming callbacks. Tools will not be able to recognize such patterns and represent them in component IDL **uses** and **provides** or **emits, publishes** and **consumes** specifications. A tool may support the designation of operations in the EJB Remote interface which are to be mapped to CORBA Component Events. However, this specification does not specify how this is to be done.

It is likely that a future version of the EJB Specification will introduce the Java Messaging Service and allow EJBs to subscribe to message channels and send and receive messages. When this version is made public, the CORBA Component specification will be revised to define interoperability with JMS.

An RMI client of an EJB can determine whether the EJB has a primary key (by examining the remote interface of the EJB Home with the Java Reflection facility), but cannot determine any other characteristics of the EJB implementation (e.g. whether is it a stateful or stateless Session Bean). This assertion is also true for CORBA Components.

11.4.2.6 CORBA Component Facade Example

In this section we show a simple EJB together with the corresponding Component IDL. Note that the EJB deployment metadata is needed to generate the IDL; this is because the metadata binds together the Remote interface and the Home interface.

Below are the remote interfaces of the EJB.

```
class CustInfo implements java.io.Serializable {
   public int custNo;
   public string custName;
   public string custAddr;
   };
class CustBal implements java.io.Serializable {
   public int custNo;
   public float acctBal;
   };
interface CustomerInquiry extends javax.ejb.EJBObject {
   CustInfo getCustInfo(int iCustNo)
      throws java.rmi.RemoteException;
   CustBal getCustBal(int iCustNo)
      throws java.rmi.RemoteException;
   };
interface CustomerInquiryHome extends javax.ejb.EJBHome {
   CustomerInquiry create()
      throws java.rmi.RemoteException;
   };
```

Below are the contents of the descriptor classes as they might be expressed in an equivalent XML document.

```
<DeploymentDescriptor>
   <versionNumber> 1 </versionNumber>
   <homeName> customer/CustomerInquiry </homeName>
   <enterpriseBeanClassName> CustomerInquiryBean
      </enterpriseBeanClassName>
   <HomeInterfaceClassName> CustomerInquiryHome
      </HomeInterfaceClassName>
   <reentrant> true </reentrant>
   <remoteInterfaceClassName> CustomerInquiry
      </remoteInterfaceClassName>
</DeploymentDescriptor>
```

The EJB is a **SessionBean**, and in this case, its **create** operation requires no parameters. The two operations take a key value and return values to the caller. The EJB implementation will use **JDBC** to retrieve the information to be returned by the operations on the **CustomerInquiry** EJB.

The serializable value classes are translated by RMI/IIOP into CORBA concrete **valuetypes** as follows:

```
valuetype CustInfo {
public:
    long custNo;
    ::CORBA::wstring custName;
    ::CORBA::wstring custAddr;
    };
valuetype CustBal {
public:
    long custNo;
    float custBal;
    };
```

The information in the deployment descriptor and the Home and remote interface declarations is introspected and used to generate the following IDL.

```
interface CustomerInquiryDefault {
    CustInfo getCustInfo(in long iCustNo);
    CustBal getCustBal(in long iCustNo);
};

component CustomerInquiry supports CustomerInquiryDefault {
    home {
        factory create();
        };
    };
```

The IDL can be used to create CORBA Component proxies that allow the use of the EJB by CORBA Components.

11.4.3 EJB Facades for CORBA Components

This specification does not address such facades. It is expected that a subsequent revision of this specification will define a standard mapping for CORBA Components into an EJB Remote and Home interface. This mapping will only be possible for some CORBA Components.

Until that time, an EJB developer wishing to use a CORBA Component may follow the guidelines listed inSection 7.5.2 on page 179 for Component Unaware Clients, assuming that his EJB implementation is using a Java CORBA proxy for the CORBA Component. If the client ORB is able to flow transactions and security information from the EJB server environment to the CORBA Component server ORB and the transaction and security services on the two servers interoperate, the resulting application should be both transactional and secure.

11.5 Enterprise Java Beans deployed to a CORBA Component Server

CORBA container providers will want to host EJBs. This non-normative section discusses strategies for hosting EJBs and presents the relationships between the EJB containment contracts and the CORBA Component Model containment contracts. The

EJB contracts with the container and other services are defined in the Enterprise JavaBeans Specification Version 1.0 and the reader is assumed to be familiar with this document.

11.5.1 EJB Hosting Strategies

The two primary strategies are:

- creating a container which provides both CORBA Component and EJB containment contracts and protocols - this is termed **direct hosting**;
- creating a set of adapter objects which make an EJB look like a CORBA Component - this is termed **EJB adaptation**.

Other strategies which mix these two basic strategies are possible.

11.5.1.1 Direct Hosting

In the direct hosting approach, the container provides both the CORBA Component and the EJB APIs. Where the ORB and container are both written in Java, deployment is simple and direct and no additional operations for interface adaptation need to be generated on the EJBObject and EJBHome. Where the ORB and container are not written in Java, the container's EJB interfaces must have Java proxies, and the EJB's interfaces must have compiled language proxies, since EJB presumes that multiple instances of the same EJB may be active with distinct identity and state.

11.5.1.2 EJB Adaptation

The EJB adaptation strategy has the unfortunate consequence of burdening the deployer with a number of artifacts to be created. In the normal case of EJB deployment, the deployer must create the **EJBObject** and the **EJBHome** implementations. For EJBs with Container Managed Persistence, one or two additional objects will need to be created.

If the ORB is a Java ORB and the container is also written in Java, an **EJBObject** and an **EJBHome** implementation can be created which adapt to the corresponding CORBA Component container interfaces. These are dispatched from the ORB skeletons created from the Java to IIOP generation process.

On the other hand, if the ORB and container are not Java, skeletons and servants must be generated in a language acceptable to the ORB and container, and these must dispatch via the JNI into the Java VM. In addition, the **EJBObject** and **EJBHome** implementations must be created in Java. There are a number of complex trade-offs about whether activation, deactivation, and other container services are done in Java or in the container implementation language. Similar trade-offs exist for persistence support of Container Managed Persistence EJBs.

11.5.2 EJBObject

The EJB Specification requires the EJB container to generate an implementation of the EJBObject. The purpose of this object is to intercept the method invocation and register transactions, make authentication and authorization checks, activate the EJB Instance and obtain its persistent state if any, and to apply other qualities of service. In a CORBA Component server, the ORB and container cooperate to create an intercept of the operation invocation. For example, all of these operations may be done by a Servant Locator which serves the purpose of the EJBObject. Thus, the CORBA Component server deployment tools for EJB deployment may generate a **ServantLocator** instead of generating an EJBObject. Once the **pre_invoke** operation on the **ServantLocator** interface has returned, the ORB will dispatch on a skeleton which directly invokes the operation on the EJB instance.

11.5.3 Transactional State Management

Stateless Session EJBs have no transactional state and consequently can be used for any operation invocation regardless of what transaction ID it may carry. Stateful Session Beans may not have transactional state and thus may not be shared by multiple clients running under different transaction IDs. Entity Beans have transactional state. To insure consistency, Entity Beans must be locked until the transaction has committed or rolled back, or the container must manage transactional copies of two Beans with the same identity that are involved in different transactions and insure that the proper Bean instance receives the method invocation. The **ServantLocator** or EJBObject interceptor allows the container the opportunity to select the proper Bean instance prior to dispatch and to check locks if necessary. Stateful Session Beans and Entity Beans are not thread safe; no more than one thread may be executing an operation in such a Bean instance at a time. It is the responsibility of the container to insure this. It is the responsibility of the Bean implementor not to create threads.

11.5.4 Container Managed Persistence

The following discussion assumes that the CORBA Component server's persistence implementation will be used to provide persistence for Entity Beans with Container Managed Persistence (CMP). A CORBA Component server may elect to provide persistence for CMP Entity Beans in any other way which is consistent with the Enterprise Java Bean specification version 1.0.

A CMP Bean has fields listed in the **EntityDescriptor** for which persistence is provided. This pattern, termed the *cached state* pattern, assumes that the persistence service sets the fields before activating the Bean and pushes the values of the fields to the datastore at the time of commit or passivation. It also requires the container to manage transactional instances of the Entity Bean.

The CORBA Component Model provides persistence using the opposite pattern, termed *delegated state*, in which the Component instance is given a reference to a Storage incarnation from which it obtains its persistent values. The CCM also supports the *cached state* pattern in that the Component instance may copy information from the Storage into its instance variables. If it does this, it becomes transactional. If it does

not, the container may manage a single instance of the Component to serve multiple transactional entities by providing the reference to the Storage incarnation at the time it dispatches the public operation on the Component instance.

A CORBA Component which caches state could use a “push” protocol or a “pull” protocol to initialize and return its cached state. However, this proposal only supports the “pull” protocol. In this protocol, the Component is responsible for implementing the caching of state during or following the invocation of the **load** operation. Likewise, it must return its cached state prior to or during the **store** operation.

The CMP bean employs the “push” protocol. In this protocol, the persistence helper (a Storage in the CCM), is expected to set the state fields and then invoke the **ejbLoad** operation.This means that a special persistence helper will be needed for EJBs. This persistence helper will look to the CORBA Component server like a Container Managed Persistent Component. During the **load** operation, the helper will retrieve the state from the Storage and set the Entity Bean’s fields. After all the fields have been set, the persistence helper will invoke **ejbLoad** on the Entity Bean instance. The EJB persistence helper is constructed during the activation protocol and given a reference to the EJB instance.

11.5.5 Bean Managed Persistence

Deploying an Entity Bean with Bean Managed Persistence will typically require the server to support **JDBC**. The CORBA Component server implementation of **JDBC** will need to be integrated with the transaction coordinator and the security service and may require a principal translation to a logon ID and password required to establish the **JDBC** connection. The JDBC 2.0 standard with the XA connection extensions provides this function.

11.5.6 EJBHome

The EJB specification assigns to the container tools the responsibility of generating a Home implementation for all EJBs. In the case of the Entity Bean with Bean Managed Persistence, the Entity Bean provides some implementation of the Home operations and the container generated Home only needs to provide transactions, security, and other qualities of service.

A CORBA Component server which wishes to host EJBs must also provide tools for the construction of a Home implementation. This Home implementation may make use of a CORBA Component Model Home Executor and its corresponding **StorageHome**, or it may use some other form. The correspondence between the EJB Home and the CCM Home is discussed inSection 11.4.2.2 on page 347.

11.5.6.1 Container Managed Persistent Entity Beans

A CORBA Component Home may manage CMP Entity Beans. Such a Home must, in conjunction with the container, create or find a persistent incarnation for the persistent fields of the Bean. It must then instantiate a CMP persistence helper as described

above and invoke the **load** operation. The CMP persistence helper will then initialize the persistent state of the CMP Bean. This activation pattern is very similar to the activation pattern for a Persistent CORBA Entity Component.

11.5.6.2 Bean Managed Persistent Entity Beans

A CORBA Component Home may manage BMP Entity Beans, but its implementation must meet the requirements for invoking the **ejbCreate*XXX*** and **ejbFind*XXX*** operations on the Bean. Container tools may generate the CORBA Component Home implementations.

11.5.6.3 Other Bean Types

A CORBA Component Home may manage other types of EJBs. The implementation of such a Home must be handcrafted from the documentation supplied with the Enterprise Java Bean.

11.5.7 Object References and Handles

EJB object references are not persistent; they are local references to a container generated object such as an **EJBObject** or an **EJBHome** implementation, or they are local references to an RMI style proxy. An EJB reference can be made persistent in two ways:

- Use **getHandle** on the remote interface to get a serializable local **Handle** class which can be made persistent by storing its serialization. The **Handle** class provides a **getObject** method which returns an object reference to the object if the object still exists and is locatable by whatever means the **Handle** provides.
- Make persistent the **JNDI** name of the Home and the Primary Key of an Entity Bean. Often this will be the state of the **Handle**, but it is not directly accessible from the **Handle**

When an EJB is presenting a CORBA Component facade and the EJB is an Entity Bean, its **oid** will contain information that allows a **Servant** to be found and the Primary Key to be extracted. Thus, persisting the object reference to the CORBA Component facade using **object_to_string** will work and be roughly equivalent to using the Home name and the Primary Key.

Should the Home of the Entity Bean be renamed, the **Handle** will be broken, but the CORBA Object Reference will continue to work so long as the POA and Servant Manager remain the same. If the POA and/or Servant Manager are changed, the CORBA Object Reference will be broken, but the **Handle** will continue to work. Administrators must be careful to change the location of Entity Beans so that so that neither kind of reference breaks.

Sometimes, it is necessary to move an Entity Bean into a new Home. The EJB specification does not define the implementation of the **Handle**, but to the greatest extent possible, **Handles** should be able to cope with such moves which may change the JNDI name of the Home.

CORBA Object References support redirection from the server. When the Entity Component is moved, the CORBA server is able to return an object reference to its new location.

If the **SessionBean** is destroyed for any reason, both the **Handle getObject** and the **string_to_object** operation will fail.

A CORBA server might implement a **Handle** using object_to_string and string_to_object.

11.5.8 EJB Context Interfaces

The CORBA Component container for hosting EJBs must provide each EJB instance with the proper context interface implementation. This will usually be a Java class which delegates to the same container code that would be executed by the corresponding CORBA component context interface. Table 11-1 on page 357 shows the correspondences between operations on these interfaces.

11.5.9 EJB Implementation Interfaces

An EJB implementation is required to implement the **EntityBean** or **SessionBean** interface. A CORBA Component container may use those interfaces just as it would the corresponding CORBA Component Interfaces. Table 11-2 on page 358 shows the correspondences between operations on these interfaces. Table 11-3 on page 359 shows the correspondences between other interfaces that may be implemented by the EJB and the CORBA Component interfaces.

11.5.10 Environment Properties

Each EJB has access to a hash table of (name, value) data items. This hash table is defined by the **DeploymentDescriptor** and should not be changed by the EJB. A single instance of the hash table is provided to each instance of a deployed EJB. A CORBA Component container may implement this by deserializing the Deployment Descriptor and extracting the hash table, or it may use some other technique.

11.5.11 JNDI and CosNaming

EJBs that have names are Homes and singletons. Developers should create EJBs which rely on fetching the name strings or components from the Environment Properties. Where this is done, the EJB developer will hopefully have documented each such EnvironmentProperty so that the deployer can set it to an appropriate value.

JNDI name strings are also used for resources, e.g. for **JDBC** database connections. These name strings do not have to be mapped to **CosNaming** names. Again, the EJB documentation must describe these cases and EJB deployment will be a manual task.

11.5.12 CORBA Component and EJB 1.0 Containment Contracts

The CORBA Component container was designed to be a superset of the EJB container. The relationship between the interfaces provided by both containers is shown in the following tables. Table 11-1 shows the relationship between the EJB context interfaces and EJB.

Table 11-1 EJB to CCM Comparison - Context Interfaces

EJB Interface	EJB Operation	CCM Interface	CCM Operations
EJBContext		ComponentContext	
	getEJBHome		get_home
	getEnvironment		Note 1
	getCallerIdentity		***Security.getCallerIdentity***
	isCallerInRole		***Security.isCallerInRole***
	getRollbackOnly		***Transaction.get_rollback_only***
	setRollbackOnly		***Transaction.set_rollback_only***
	getUserTransaction		get_transaction
			get_reference
			get_home_registration
			get_security
			get_events
SessionContext		TransientContext	
	getEJBObject		get_reference (Note 2)
			get_transient_origin
EntityContext		PersistentContext	
	getEJBObject		get_reference (Note 2)
	getPrimaryKey		***Storage.get_primary_key***
			get_component_id
			get_persistent_origin
			get_storage

1. The EJB environment properties list is accomplished using the configurator mechanism for CORB components. Consequently, the **get_environment** operation is not mapped.

2. The **get_reference** operation is available by inheritance from **ComponentContext**.

Table 11-2 below compares the EJB callback interfaces with their CORBA components counterparts.

Table 11-2 EJB to CCM Comparison - Callback Interfaces

EJB Interface	EJB Operation	CCM Interface	CCM Operations
EnterpriseBean		EnterpriseComponent	
		ServiceComponent	
			set_transient_context
SessionBean		SessionComponent	
	setSessionContext		set_transient_context (Note 3)
	ejbRemove		remove
	ejbActivate		activate
	ejbPassivate		passivate
EntityBean		EntityComponent	
	setEntityContext		set_persistent_context
	unsetEntityContext		unset_persistent_context
	ejbRemove		remove
	ejbActivate		activate
	ejbPassivate		passivate
	ejbLoad		load
	ejbStore		store
SessionSynchronization		Synchronization	
	afterBegin		Note 4
	beforeCompletion		before_completion
	afterCompletion		after_completion

3. The **set_transient_context** operation is supported by inheritance from **ServiceComponent**.

4. OTS does not provide this level of notification and it is not supported by many transaction managers

Table 11-3 below compares the other EJB internal interfaces to the CORBA component equivalents.

Table 11-3 EJB to CCM Comparison - Other Internal Interfaces

EJB Interface	EJB Operation	CCM Interface	CCM Operations
UserTransaction		Transaction	
	begin		begin
	commit		commit
	rollback		rollback
			get_rollback_only
	setRollbackOnly		set_rollback_only
	getStatus		get_status
	setTimeout		set_timeout
	JTS.Current.suspend		suspend
	JTS.Current.resume		resume
		Security	
	EJBContext.getCallerIdentity		get_caller_identity
	EJBContext.isCallerInRole		is_caller_in_role
		Storage	
	EntityContext.getPrimaryKey		get_primary_key

11.5.13 Deployment Processes and Artifacts

In principle, EJB deployment tools for a CORBA Component Server could first examine the EJB jar file and produce CIDL. The CIDL to implementation translator could then be invoked to generate the necessary pieces to deploy the EJB. Some parts of these generated pieces must be completed by hand (e.g. the Home implementation). Since the parts are intended to adapt the CORBA Component container and services to the EJB, they are not the same as the usual parts that are generated from the CIDL translator. Consequently a command line flag or some other device will be needed to cause the CIDL translator to emit parts intended to adapt to EJBs. Some additional metadata not found in the CIDL will also be needed (to described the EJB interfaces).

It is simpler to have the EJB deployment tools produce the implementation artifacts that the CIDL translator would produce. This avoids an extra step in the development process and the need for a command line flag on the CIDL translator.

C++ Language Mapping 12

12.1 Introduction

This chapter specifies the C++ language mappings for the CIDL **storage** construct.

12.2 Mapping for incarnations

An incarnation is a manifestation of a storage type in an excecution context. As such, the language mapping for a storage type is actually the language mapping for the incarnation. Throughout this chapter we will generally refer to a storage type in terms of its *incarnation*, since language mappings necessarily pertain to incarnations.

The mapping for incarnation distinguishes between *dependent* and *independent* incarnations, as defined in Section 6.7.9 on page 107. An incarnation maps to the following C++ classes:

- an abstract state class that provides pure virtual member functions for storage member accessors and mutators, but no operations relating to storage object identity or life cycle. This abstract state class is used as the type of dependent members.
- an independent incarnation class that inherits both the storage type's abstract state class and the **Components::Persistence::IndependentBase** class that provides identity and life cycle management member functions.

The independent incarnation class is an abstract base class that has the same name as the CIDL **storage** definition. CIF implementations are responsible for creating incarnations that are instances of concrete classes derived from either the abstract state class (for incarnations of dependent members) or the independent incarnation class (for independent incarnations). The concrete classes that the CIF instantiates are not exposed to applications. Applications obtain instances of incarnations from storage homes, and from other incarnations.

Applications manage incarnations via actual C++ pointers.

Because storage object semantics support the sharing indepdent incarnations within graphs of other incarnations, the lifetimes of C++ independent incarnations are managed via reference counting. Reference counting operations for C++ incarnations are directly implemented by those instances. CIF implementor are responsible for providing implementations of reference counting on incarnations.

As for most other types in the C++ mapping, each incarnation type also has an associated C++ **`_var`** type that automates its reference counting.

For a storage type in the following form:

// CIDL
storage *<storage_name>* { ... };

The C++ mapping defines the following classes:

```
// C++
class <storage_name>AbstractState
: public virtual Components::Persistence::IncarnationBase
{ ... };

class <storage_name>
: public virtual
<storage_name>AbstractState,
Components::Persistence::IndependentBase
{ ... };
```

Storage type inheritance corresponds to inheritance of abstract state classes. For storage definitions of the following forms:

// CIDL
storage *<base_name>* { ... };

storage *<derived_name>* : *<base_name>* { ... };

The C++ mapping defines the following classes:

```
// C++
class <base_name>AbstractState
: public virtual Components::Persistence::IncarnationBase
{ ... };

class <base_name>
: public virtual
<base_name>AbstractState,
Components::Persistence::IndependentBase
{ ... };

class <derived_name>AbstractState
: public virtual <base_name>AbstractState
{ ... };

class <derived_name>
: public virtual
<derived_name>AbstractState,
Components::Persistence::IndependentBase
{ ... };
```

12.2.1 Incarnation members

For the purposes of this discussion, the terms *accessors*, *modifiers*, and *referents* are being used in precisely the same manner as the C++ language mapping specification, to refer to functions that provide read-only access, read-write access, and write access to state encapsulated by a class, respectively.

12.2.1.1 Atomic members

The C++ mapping for atomic members of incarnations is essentially identical in form to the current C++ mapping for public state members of value types. Storage members are mapped to public pure virtual accessor, modifier and (in some cases) referent functions on the C++ incarnation abstract state class, precisely as they would for public members of the same type in value type mappings.

Atomic member acessor and modifier functions on incarnations behave precisely as do the corresponding functions on C++ value types. Atomic member reference functions behave somewhat differently.

The implementation of the incarnation maintains the notion of an *internal logical state*, which consists of the initial state at the time of incarnation or the most recent refresh operation, plus the application of all modifier function invocations made in the current transactional context. Internal logical state is maintained an a per-transactional-context basis. Modifications made to values obtained through reference functions do not modify the internal logical state of the incarnation. The internal logical state must be maintained so that it does not “see” changes until they are explicitly changed via modifier functions. The following behaviors must be enforced:

- Values returned by read-only accessors shall return the current internal logical state.

- Objects returned by a reference function shall be shared among all threads that belong to the same transactional context. Modifications made to those objects will be visible only within the transaction context, and they will not be reflected in the incarnation's internal logical state.
- The internal logical state may only be modified by modifier functions.

Consistent application of these constraints requires the addition of a read-only accessor function for value types. Given the following CIDL:

```
// CIDL
valuetype <value_name> { ... };
storage <storage_name> {
<value_name> <value_member>;
};
```

the mapping for the value type member would be as follows:

```
// C++

class <value_type_name> : public virtual ValueBase { ... };

class <storage_name>AbstractState {
 public:
   ...
virtual const <value_name>*
<value_member> () const = 0;     // 1
virtual <value_name>*
<value_member>Shared () const = 0; // 2
virtual void
<value_member> (const <value_name>*) = 0; // 3
};
```

The const accessor on line 1 returns the value of the internal logical state of the incarnation. The non-const referent on line 2 returns the transient value shared by all threads in the same transactional context. The modifier in line 3 modifies the internal logical state.

12.2.1.2 Independent storage members

Independent storage members map to an pair of accessor and mutator functions whose signatures use the independent incarnation type (i.e., the type that inherits from IndependentBase). Given a storage definition of the following form:

```
// CIDL
storage <storage_name> {
strong <storage_name> <member_name>;
};
```

the mapping for the independent incarnation member would be as follows:

```
// C++
class <storage_name>;
class <storage_name>AbstractState : public virtual
Components::Persistence::IncarnationBase{
  public:
...
virtual <storage_name>*
<member_name> () const = 0;

virtual void
<member_name> (<storage_name>*) = 0;
};

class <storage_name> : public virtual
<storage_name>AbstractState,
Components::Persistence::IndependentBase
{ ... };
```

The internal logical state of the incarnation that defines an independent member holds a reference for the member, which is a distinct storage object. The internal logical value of an independent member is, in essence, the PID of the storage object assigned by the member's modifier.

The accessor function does not increment the reference count of the returned incarnation. This implies that the caller of the accessor does not adopt the return value. The modifier function increments the reference count of its argument, then decrements the reference count of the member incarnation it is replacing before returning.

12.2.1.3 Storage sequence members

As defined in Section 6.8.3 on page 118, independent members may specify a type name that denotes a sequence of storage types. As described there, the independent characteristic of the member declaration and the nature of the reference (**strong** or **weak**) apply to the elements of the sequence, not the sequence itself. In the C++ language mapping, incarnation members that are sequences of storage members behave much like dependent members, as follows:

- The incarnation that owns the sequence member maintains the sequence's state (i.e., the set of references to independent incarnations that constitutes the sequence) as part of its internal logical state.
- The owning incarnation creates and manages the actual instance of the sequence class that it exposes through the accessor to the sequence member. This C++ object must be the same object throughout the life time of the incarnation.
- The modifier for the sequence member copies the state of its argument into the sequence maintained by the owning incarnation.

- The sequence class that is created an maintained by the incarnation for storage sequence members has a programming interface that is a subset of the normal sequence mapping. The class defined by this mappings is an abstract base class. The CIF provider is free to define and construct any concrete class that provides the required semantics

Given a CIDL definition of the following form:

```
// CIDL
storage <storage_name> {
strong sequence < <storage_name> > <member_name>;
};
```

the mapping for the storage sequence member would be as follows:

```
// C++
class <storage_name>;

class <storage_name>Sequence {
public:
void length(ULong) = 0;
ULong length() const = 0;
<storage_name>* _element_at(ULong index) const;
void _set_element_at(<storage_name>*, ULong index);

protected:
<storage_name>Sequence();
virtual ~<storage_name>Sequence();

private:
<storage_name>Sequence(const<storage_name>Sequence&);
void operator=(const <storage_name>Sequence&);
};

class <storage_name>AbstractState {
public:
...
virtual <storage_name>Sequence*
<member_name> () const = 0;
virtual void
<member_name> (<storage_name>Sequence*) = 0;
virtual void
<member_name> (ULong length, <storage_name>**) = 0;
...
};

class <storage_name> : public virtual
<storage_name>AbstractState,
Components::Persistence::IncarnationBase
{ ... };
```

12.2.1.4 Dependent members

Dependent members map to a pair of accessor and modifier functions whose signatures contain the abstract state classes for the specified storage type. Given a storage definition of the following form:

storage *<storage_name>* {
***<storage_name> <member_name>*;**
};

the mapping for the dependent incarnation member would be as follows:

```
// C++

class <storage_name>AbstractState {
 public:
...
virtual <storage_name>AbstractState* <member_name>
() const = 0;
virtual void <member_name> (
<storage_name>AbstractState*) = 0;
};
```

The internal logical state of the incarnation that defines the dependent member maintains the dependent member by value, so that the values of the members of the dependent member are incorporated into its internal logical state. The modifier for a dependent member performs a deep copy of its argument. Note that an independent incarnation can be passed as the argument to a dependent member modifier. The value of the independent member is deep-copied into the value of the dependent member.

The accessor of a dependent member is never nil. Dependent members of newly-created incarnations have intial values as described in Section 6.8.3 on page 118. The pointer returned by a a dependent member's accessor shall be the same pointer for the life time of incarnation that owns the dependent member.

For example, assume the follow storage definition:

```
// CIDL
storage A {
     string name;
     long num;
};
storage B {
     A depA;
     strong A indepA;
};

// C++
// assume existence of BStorageHome
B* incarnB = BStorageHome->create();

incarnB->depA()->num(100);
// this is the preferred usage;
// the value of num is incorporated into
// the internal logical tate of incarnB

A_AbstractState* depIncarnA = incarnB->depA();
depIncarnA->name("jeff");
// less obvious usage;
// the value "jeff" is incorporated into
// the internal logical state of incarnB
// in this case, too; all of the members
// of depIncarnA are part of incarnB's
// internal logical state

A* incarnA = AStorageHome->create();
```

```
incarnA->name("geoff");
incarnA->num(-100);
incarnB->indepA(incarnA);
// incarnB->indepA holds a reference
// to incarnA

incarnB->depA(incarnA);
// the values from incarnA are copied
// into the members of depA in incarnB's
// internal logical state
```

12.2.2 Constructors, Assignment Operators, and Destructors

A C++ incarnation class defines a protected default constructor and a protected virtual destructor. The default constructor is protected to allow only derived class instances to invoke it, while the destructor is protected to prevent applications from invoking delete on pointers to incarnation instances instead of using reference counting operations. The destructor is virtual to provide for proper destruction of derived incarnation class instances when their reference counts drop to zero.

Portable applications shall not invoke an incarnation class copy constructor or default assignment operator. Due to the required reference counting, the default assignment operator for a storage type class shall be private and preferably unimplemented to completely disallow assignment of storage type instances.

12.2.3 _downcast operation

A static **`_downcast`** function is provided by each incarnation class to provide a portable way for applications to cast down the C++ inheritance hierarchy. If a null pointer is passed to **`_downcast`**, it returns a null pointer. Otherwise, if the incarnation pointed to by the argument is an instance of the incarnation class being downcast to, a pointer to the downcast-to class type is returned. If, however, the incarnation pointed to by the argument is *not* an instance of the incarnation class being downcast to, a null pointer is returned.

12.2.4 _type_id operation

A static **`_type_id`** operation is provided by each independent incarnation class. This operation returns the CIDL type identifier of the storage type definition corresponding to the incarnation class. CIDL type identifiers are described in Section 4.3.1 on page 27.

12.2.5 Example

For example, consider the following CIDL **storage** definition:

```
// CIDL
storage example {
short snum;
long lnum;
string str;
float fnum;
example dep;
strong example indep;
};
```

The C++ mapping for incarnations of this storage type is:

```
// C++
class exampleAbstractState
: public virtual Components::Persistence::IncarnationBase {
  public:

virtual Short snum() const = 0;
virtual void snum(Short) = 0;

virtual Long lnum() const = 0;
virtual void lnum(Long) = 0;

virtual const char* str() const = 0;
virtual void str(const char*) = 0;
virtual void str(const String_var&) = 0;

virtual Float fnum() const = 0;
virtual void fnum(Float) = 0;

virtual exampleAbstractState* dep() const = 0;
virtual void dep(exampleAbstractState*) = 0;

virtual example* indep() const = 0;
virtual void indep(example*) = 0;

static exampleAbstractState*_downcast(
IncarnationBase*);

protected:
exampleAbstractState();
virtual ~exampleAbstractState();
private:
// private and unimplemented
void operator=(const exampleAbstractState&);
};


// C++
class example
: public virtual exampleAbstractState,
Components::Persistence::IndependentBase {
  public:
static example*_downcast(IncarnationBase*);
static CORBA::RepositoryId _type_id();

protected:
example();
virtual ~example();
private:
// private and unimplemented
void operator=(const example&);
};
```

12.2.6 IncarnationBase

The **IncarnationBase** class serves as an abstract base class for all incarnations, both dependent and independent. The primary purpose of the **IncarnationBase** class is to support downcasting.

```
// C++
namespace Components {
namespace Persistence{

class IncarnationBase {
  public:
static IncarnationBase* _downcast(IncarnationBase*);

  protected:
IncarnationBase();
IncarnationBase(const IncarnationBase&);
virtual ~IncarnationBase();

  private:
void operator=(const IncarnationBase&);
};
};
};
```

12.2.7 IndependentBase and reference counting

The **IndependentBase** class serves as an abstract base class for all C++ indepdent incarnation classes. **IndependentBase** provides several pure virtual reference counting functions inherited by all independent incarnation classes:

```
// C++
namespace Components {
namespace Persistence {

class IndependentBase {
  public:
virtual PersistentId _get_pid()=0;
virtual PersistentStoreBase*
_get_persistent_store()=0;
virtual StorageHomeBase* _get_storage_home()=0;

virtual void _flush()=0;
virtual void _refresh()=0;

virtual _add_ref() = 0;
virtual _remove_ref() = 0;

virtual ULong _refcount_value() = 0;

  protected:
IndependentBase();
IndependentBase(const IndependentBase&);
virtual ~IndependentBase();

  private:
void operator=(const IndependentBase&);
};
};
```

_get_pid

This function returns the persistent ID of the storage object which the incarnation represents.

_get_persistent_store

This function returns a pointer to the persistent store object that created and manages the incarnation.

_get_storage_home

This function returns a pointer to the storage home object that created and manages the incarnation.

_flush

This function synchronizes the state of the incarnation with the underlying store.

_refresh

This function refreshes the state of the incarnation with that of the underlying store.

_add_ref

This function increments the reference count of an incarnation.

_remove_ref

This function decrements the reference count of an incarnation and deletes the incarnation when the reference count drops to zero. Note that the use of **delete** to destroy instances requires that all incarnations be allocated using **new.**

_refcount_value

This function returns the value of the reference count for the incarnation on which it is invoked.

The names of these functions begin with underscore to keep them from clashing with user-defined operations in derived incarnation classes.

IndependentBase also provides a protected default constructor, a protected copy constructor, and a protected virtual destructor. The copy constructor is protected to disallow copy construction of derived incarnations except from within derived class functions, and the destructor is protected to prevent direct deletion of instances of classes derived from **IndependentBase**.

With respect to reference counting, **IndependentBase** is intended to introduce only the reference counting interface. Depending upon the inheritance hierarchy of a incarnation class, its instances may require different reference counting mechanisms.

Java Language Mapping 13

13.1 Introduction

This section specifies the Java language mappings of the CIDL **storage** construct.

13.2 Mapping for incarnations

An incarnation is a manifestation of a storage type in an excecution context. As such, the language mapping for a storage type is actually the language mapping for the incarnation. Throughout this chapter we will generally refer to a storage type in terms of its *incarnation*, since language mappings necessarily pertain to incarnations.

The mapping for incarnation distinguishes between *dependent* and *independent* incarnations, as defined in Section 6.7.10.6 on page 110 and. An incarnation maps to the following Java interfaces:

- an *abstract state interface* that provides accessor and mutator methods for the members of the storage type, but no methods relating to storage object identity or life cycle. This abstract state class is used as the type of dependent members. Abstract state interfaces extend the **`org.omg.Components.Persistence.IncarnationBase`** interface.
- an *independent incarnation interface* that inherits both the storage type's abstract state interface and the **`org.omg.Components.Persistence.IndependentBase`** interface that provides identity and life cycle management methods.

The independent incarnation interface has the same name as the CIDL **storage** definition. CIF implementations are responsible for creating incarnations that are instances of concrete classes that implement either the abstract state interface (for incarnations of dependent members) or the independent incarnation interface (for independent incarnations). The concrete classes that the CIF instantiates are not exposed to applications. Applications obtain instances of incarnations from storage homes, and from other incarnations.

For a storage type in the following form:

```
// CIDL
storage <storage_name> { ... };
```

The Java mapping defines the following classes:

```
// Java
interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{ ... }

interface <storage_name> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{ ... }

public class <storage_name>Helper {
public static String _type_id();
}
```

Storage type inheritance corresponds to inheritance of abstract state interfaces. For storage definitions of the following forms:

```
// CIDL
storage <base_name> { ... };

storage <derived_name> : <base_name> { ... };
```

The Java mapping defines the following interfaces:

```
// java
public interface <base_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{ ... };

public interface <base_name> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{ ... };

public class <base_name>Helper {
public static String _type_id();
}

public interface <derived_name>_AbstractState extends
<base_name>AbstractState,
org.omg.Components.Persistence.IncarnationBase
{ ... };

public interface <derived_name> extends
<derived_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{ ... };

public class <derived_name>Helper {
public static String _type_id();
}
```

13.3 Incarnation members

In general, members map to accessor and mutor methods on the abstract state interface of the storage type. There are minor differences in how certain members map, depending on whether the members are independent or dependent members, or atomic members, or whether the type of the member maps to a Java primitive or reference type.

13.3.1 Atomic members

Atomic members whose IDL/CIDL types map to primitive (i.e., non-reference) Java types map to an accessor method and a mutator method. Both methods have the same name as the member in the storage definition. The accessor returns the mapped member type, and the mutator returns void, taking a single parameter of the mapped member type.

13.3.1.1 Primitive types

Given a storage type with an atomic member that maps to a Java primitive type, in the form:

storage <storage_name> {
<member_type> <member_name>;
};

The resulting Java abstract state interface would have the following form:

```
// Java
public interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{
public <mapped_member_type> <member_name> ();
public void <member_name> ( <mapped_member_type> );
}

public interface <storage_name> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{}

public class <storage_name>Helper {
public static String _type_id();
}
```

13.3.1.2 Reference types

The implementation of the incarnation maintains the notion of an *internal logical state*, which consists of the initial state at the time of incarnation or the most recent refresh operation, plus the application of all modifier function invocations made in the current transactional context. Internal logical state is maintained an a per-transactional-context basis. Maintaining a consistent internal logical state is problematic when the type of the member is a reference type. Performing a deep copy of the value on every access may be extremely inefficient. The mapping for atomic members of reference types represents a compromise between programming simplicity and efficiency.

Atomic members whose types map to reference Java types map to three methods, a *copy accessor* method, a *reference accessor* method, and a *mutator* method. The copy accessor method returns a deep copy of the internal logical state of the member. The reference accessor returns a reference to a transient copy of the member's state. The mutator method sets the internal logical state of the incarnation.

Modifications made to instances of reference types obtained from reference accessors do not modify the internal logical state of the incarnation. The internal logical state must be maintained so that it does not "see" changes until they are explicitly changed via mutator functions. The following behaviors must be enforced:

- Objects returned by copy accessors shall return a deep copy of the current internal logical state of the member.

- Objects returned by reference accessors shall be shared among all threads that belong to the same transactional context. Modifications made to those objects will be visible only within the transaction context, and they will not be reflected in the incarnation's internal logical state.
- The internal logical state may only be modified by mutators.

Given a storage type with an atomic member that maps to a Java reference type, in the form:

storage <storage_name> {
<member_type> <member_name>;
};

The resulting Java abstract state interface would have the following form:

```
// Java
public interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{
// copy accessor
public <mapped_member_type> <member_name> ();

// reference accessor name begins with underscore
public <mapped_member_type> _<member_name> ();

// mutator
public void <member_name> ( <mapped_member_type> );
}

public interface <storage_name> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{}

public class <storage_name>Helper {
public static String _type_id();
}
```

13.3.2 Independent storage members

Independent storage members map to an pair of accessor and mutator methods whose signatures use the independent incarnation type (i.e., the type that inherits from IndependentBase). Given a storage definition of the following form:

// CIDL
storage *<storage_name>* {
strong *<storage_name> <member_name>*;
};

The resulting Java abstract state interface would have the following form:

```
// Java
public interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{
public <storage_name> <member_name> ();
public void <member_name> ( <storage_name> );
}

public interface <storage_name> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{}

public class <storage_name>Helper {
public static String _type_id();
}
```

The internal logical state of the incarnation that defines an independent member holds a reference for the member, which is a distinct storage object. The internal logical value of an independent member is, in essence, the PID of the storage object assigned by the member's mutator.

13.3.3 Storage sequence members

As defined in Section 6.7.5 on page 104, independent members may specify a type name that denotes a sequence of storage types. As described there, the independent characteristic of the member declaration and the nature of the reference (**strong** or **weak**) apply to the elements of the sequence, not the sequence itself. In theJava mapping, incarnation members that are sequences of storage members behave much like dependent members, as follows:

- The incarnation that owns the sequence member maintains the sequence's state (i.e., the set of references to independent incarnations that constitutes the sequence) as part of its internal logical state.
- Storage sequences in the Java mapping are represented by interfaces that provide the same basic semantics as an array.
- The owning incarnation creates and manages the actual instance of the sequence class that it exposes through the accessor to the sequence member. This object must be the same object throughout the life time of the incarnation.
- The element mutator on the sequence interface copies the state of its argument into the sequence maintained by the owning incarnation, i.e., the internal logical state of the incarnation.
- The CIF provider is free to define and construct any concrete class that supports the appropriate sequence interface and provides the required semantics

13.3.3.1 Sequence interface

A sequence of a storage type is mapped to a Java collection interface whose name formed by the name of the storage class with the string “Sequence” appended. This interface is defined in the same package as the abstract state interface of the storage type contained in the sequence. This interface provides :

- an accessor to get an individual element
- a mutator to modify an individual element
- an accessor to return the current length

The semantics of the collection class is designed to be analogous to the semantics of Java arrays. An implementation of the class shall implement the semantics specified below in any manner it chooses:

- The **setElement()** method sets the specified element. Standard Java bounds checking is performed and standard Java runtime exceptions thrown as appropriate if the position is out of bounds.
- The **elementAt()** method returns the specified element. Standard Java bounds checking is performed and standard Java runtime exceptions thrown as appropriate if the position is out of bounds

It is the responsibility of the CIF implementation to persist sequence entries that have been touched by the application.

A sequence of Storagetype of the following form:

```
// CIDL


storage <element_type> {
...
};

storage <storage_name> {
strong sequence< <element_type> > <member_name>;
};
```

is mapped to:

```
// Java
interface <element_type>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{ ... }

interface <element_type> extends
<storage_name>AbstractState,
org.omg.Components.Persistence.IndependentBase
{ ... }

public class <element_type>Helper {
public static String _type_id();
}

public interface <element_type>Sequence {
public int length();
public <element_type> elementAt(int pos);
public void setElementAt(int pos,<element_type> element);
}

public interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{
public <element_name>Sequence <member_name> ();
public void <member_name> (<element_name>Sequence seq);
public void <member_name> (<element_name>[] array);
}
```

The member mutator that takes the sequence parameter copies the contents of the parameter into the sequence maintained by the incarnation. The mutator that takes the array parameter copies the contents of the array into the sequence maintained by the incarnation.

13.3.4 Dependent members

Dependent members map to a pair of accessor and modifier functions whose signatures contain the abstract state classes for the specified storage type. Given a storage definition of the following form:

```
// CIDL
storage <storage_name> {
<storage_name> <member_name>;
};
```

The resulting Java abstract state interface with a dependent member would have the following form:

```
// Java
public interface <storage_name>AbstractState extends
org.omg.Components.Persistence.IncarnationBase
{
public <storage_name>AbstractState <member_name> ();
public void <member_name>(<storage_name>AbstractState s);
}
```

The internal logical state of the incarnation that defines the dependent member maintains the dependent member by value, so that the values of the members of the dependent member are incorporated into its internal logical state. The mutator for a dependent member performs a deep copy of its argument. Note that an independent incarnation can be passed as the argument to a dependent member mutator. The value of the independent member is deep-copied into the value of the dependent member.

The accessor of a dependent member is never nil. Dependent members of newly-created incarnations have intial values as described in Section 6.8.3 on page 118. The pointer returned by a a dependent member's accessor shall be the same pointer for the life time of incarnation that owns the dependent member.

For example, assume the follow storage definition:

```
// CIDL
storage A {
    string name;
    long num;
};
storage B {
    A depA;
    strong A indepA;
};

// Java
// assume existence of BStorageHome
B incarnB = BStorageHome.create();

incarnB,depA().num(100);
// this is the preferred usage;
// the value of num is incorporated into
// the internal logical tate of incarnB

AAbstractState depIncarnA = incarnB.depA();
depIncarnA.name("jeff");
// less obvious usage;
// the value "jeff" is incorporated into
// the internal logical state of incarnB
// in this case, too; all of the members
// of depIncarnA are part of incarnB's
// internal logical state

A incarnA = AStorageHome.create();
incarnA.name("geoff");
incarnA.num(-100);
incarnB.indepA(incarnA);
// incarnB.indepA holds a reference
```

```
// to incarnA

incarnB.depA(incarnA);
// the values from incarnA are copied
// into the members of depA in incarnB's
// internal logical state
```

13.3.5 IncarnationBase

The **IncarnationBase** interface serves as an base interface for all incarnations, both dependent and independent. **IncarnationBase** is empty.

```
// Java
package org.omg.Components.Persistence;

public interface IncarnationBase { }
```

13.3.6 IndependentBase

The **IndependentBase** class serves as an base interface for all indepdent incarnation classes.

```
// Java
package org.omg.Components.Persistence;

public interface IndependentBase {
public PersistentId _get_pid();
public PersistentStoreBase _get_persistent_store();
public StorageHomeBase _get_storage_home();

public void _flush();
public void _refresh();
};
};
```

_get_pid

This method returns the persistent ID of the storage object which the incarnation represents.

_get_persistent_store

This function returns a reference to the persistent store object that created and manages the incarnation.

_get_storage_home

This function returns a reference to the storage home object that created and manages the incarnation.

_flush

This function synchronizes the state of the incarnation with the underlying store.

_refresh

This function refreshes the state of the incarnation with that of the underlying store.

Changes to CORBA and Services 14

This chapter will provide instructions for the OMG editors as to where the new material which supports CORBA components will be placed in the existing OMG specifications.

14.1 Changes to the CORBA Core

Known contents includes:

- New resolve_initial_references ObjectID
- changes to CORBA::Object,
- new abstract meta-class CORBA::Component
- changes to the Interface Repository
- IDL changes
- **local** interfaces

14.1.1 Changes to the ORB interface

This specification adds the **Components::HomeFinder** to the list of initial references supported by the ORB. This reference is obtained using a new **ObjectID, "ComponentHomeFinder**" with **CORBA::ORB::resolve_initial_references**. The client uses this operation to obtain a reference to the **HomeFinder** interface. This requires the following enhancement to the **ORB** interface definition:

```
module CORBA {

        interface ORB {
        Object resolve_initial_references (in ObjectID identifier)
            raises (InvalidName);
        };
};
```

The string, **"ComponentHomeFinder"** is added to the list of valid **ObjectID** values.

The **HomeFinder** interface allows the client to obtain the home that creates components of a specific type.

14.1.2 Changes to the Object interface

The CORBA component specification extends the **CORBA::Object** pseudo interface with a single operation:

```
module CORBA {
    interface Object { // PIDL
        ...
        Object get_component ( );
    };
};
```

If the target object reference is itself a component reference (i.e., it denotes the component itself), the **get_component** operation returns the same reference (or another equivalent reference). If the target object reference is a facet reference the **get_component** operation returns an object reference for the component. If the target reference is neither a component reference nor a provided reference, **get_component** returns a nil reference.

Implementation of get_component

As with other operations on **CORBA::Object**, **get_component** is implemented as a request to the target object. Following the pattern of other **CORBA::Object** operations (i.e., **_interface**, **_is_a**, and **_non_existent**; see section 15.4.1.2 of the CORBA 2.3 specification), the operation name in GIOP request corresponding to **get_component** shall be "**_component**".

Programming skeletons generated by the Component Implementation Framework for components and facets shall provide standard implementations for **get_component** (i.e., the **_component** request).

14.1.3 Local interface types

This specification provides a new CORBA meta-type that is used to define programming interfaces for locality-constrained objects. The syntax is similar to that of CORBA object interfaces, but the resulting type cannot be marshaled or remotely invoked. The local meta-type is intended to obviate the need for PIDL, to obviate the

need for defining special “locality-constrained” cases of CORBA interfaces or abstract value types, and to provide users with a language-independent mechanism for declaring programming interfaces on local objects that leverages the CORBA typing system.

The grammar for specifying local interfaces is defined by the following BNF:

<local> ::= <local_header> “{“ <local_member>* “}”

<local_header> ::= “local” [<local_inheritance_spec>]

<local_inheritance_spec> ::= “:” <local_name>
{ “,” <local_name> } *

<local_member> ::= <local_op_dcl>
| <attr_dcl>
| <type_dcl>
| <const_dcl>
| <except_dcl>

<local_op_dcl> ::= <op_type_spec> <identifier> <parameter_dcls>
[<raises_expr>]

<local_name> ::= <scoped_name>

<local_base_type> ::= “localBase”

The semantics associated with local types are as follows:

- Local types cannot be marshaled. Consequently, local types (including sequences and arrays of local types) may not appear as parameters (or as components of any types that appear as parameters) of operations on CORBA Object interfaces. Local types (including sequences and arrays of local types) may not be members of structs, unions, or valuetypes. Local types may not be inserted into values of type **any**.
- Local types may appear as parameters or return values of operations on local types, or as attributes on local types.
- Parameters and return values of operations on local types may be any CORBA type. Attributes on local types may be any CORBA type.
- Language mappings for local types shall consist of the minimal language construct that satisfies the requirements of local types. In most object-oriented languages, it is expected that local types will be mapped to the language’s fundamental object type, if one exists. The semantics of invocations on local types are the semantics of function or method calls in the underlying programming languages.
- When possible, language mappings for local types shall be syntactically similar to the mappings for interfaces. Inasmuch as possible, invocations on local types shall be consistent syntactically with invocations on CORBA objects with similar signatures.

- Language mappings shall specify the form of skeletons for local types to be generated by ORB products, allowing ORB users to provide implementations of local types. There is no specified generalized framework for managing the life cycles of user-defined local types (e.g., no standard factory mechanism). The life cycles of user-defined local types are determined by the life cycle constructs of underlying programming languages for base object types (e.g., constructors/destructors, garbage collection. etc.)
- Instances of local types have no inherent identities beyond their identities as programming objects. Specifically, there is no support for the concept of a reference to a local type, other than the basic programming language construct for referring to objects.
- Instances of local types defined as part of OMG specifications to be supplied by ORB products or object service products shall be exposed through the **ORB::resolve_local** operation or through some other local object obtained from **resolve_local**.
- The **localBase** keyword denotes the generalization of local types. When **localBase** is the formal type of a parameter in an operation, an instance of any specific local type may be passed as the actual parameter.
- Local types cannot be mapped to asynchronous invocation forms as specified by the CORBA Messaging Service specification.

14.1.4 resolve_local

This specification defines a new operation on the ORB pseudo-object that allows application programmers to obtain services expressed as local types. It is similar to **ORB::resolve_initial_references**, except that the operation return value is type **localBase**. The PIDL definition is as follows:

```
module CORBA {
// PIDL
interface ORB {
localBase resolve_local(in string name)
raises (InvalidName);
};
};
```

The string parameter to the **resolve_local** operation denotes a specific local object that is managed and supplied by the ORB or by services cooperating with the ORB. Specifications that define local interfaces that are not implemented by applications shall specify unique strings that will denote well-known local objects that can be obtained from **resolve_local**.

14.1.5 Import

This specification extends IDL to provide a mechanism for importing external name scopes into IDL specifications.

The grammar for the import statement is described by the following BNF:

<specification> ::= <import>* <definition>+

<import> ::= “import” <imported_scope> “;”

<imported_scope> ::= <scoped_name> | <string_literal>

The <imported_scope> non-terminal may be either a fully-qualified scoped name denoting an IDL name scope, or a string containing the interface repository ID of an IDL name scope, i.e., a definition object in the repository whose interface derives from **CORBA::Container**.

The definition of import obviates the need to define the meaning of IDL constructs in terms of “file scopes”. This specification defines the concepts of a specification as a unit of IDL expression. In the abstract, a specification consists of a finite sequence of ISO Latin-1 characters that form a legal IDL sentence. The physical representation of the specification is of no consequence to the definition of IDL, though it is generally associated with a file in practice.

Any scoped name that begins with the scope token (“**::**”) is resolved relative to the global scope of the specification in which it is defined. In isolation, the scope token represents the scope of the specification in which it occurs.

A specification that imports name scopes must be interpreted in the context a well-defined set of IDL specifications whose union constitutes the space from within which name scopes are imported. By “a well-defined set of IDL specifications”, we mean any identifiable representation of IDL specifications, such as an interface repository. The specific representation from which name scopes are imported is not specified, nor is the means by which importing is implemented, nor is the means by which a particular set of IDL specifications (such as an interface repository) is associated with the context in which the importing specification is to be interpreted.

The effects of an import statement are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in IDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- IDL module definitions may re-open modules defined in imported name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.
- When an name scope is imported, the names of the enclosing scopes in the fully-qualified pathname of the enclosing scope are *exposed* within the context of the importing specification, but their contents are not imported. An importing specification may not re-define or re-open a name scope which has been exposed (but not imported) by an import statement.

- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this specification, name scopes that can be imported (i.e., specified in an import statement) include the following: modules, interfaces, valuetypes, structures, unions, and exceptions.
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This specification does not define a particular form for generated stubs and skeletons in any given programming language. In particular, it does not imply any normative relationship between units specification and units of generation and/or compilation for any language mapping.

14.1.6 Repository identity declarations

This specification defines extensions to IDL to allow repository identifier values to be declared in a portable, standard manner. This mechanism is intended to obviate the **#pragma** mechanism currently specified (speaking in approximate terms) in section 10.6, “RepositoryIds”, of the CORBA 2.3 specification. Should this specification be adopted, the **#pragma** mechanisms shall be deprecated.

The following grammatical productions shall be added to the IDL grammar:

<type_id_dcl> ::= “typeId” <scoped_name> <string_literal>

<type_prefix_dcl> ::= “typePrefix” <scoped_name> <string_literal>

The syntax of a repository identity declaration is as follows:

<type_id_dcl> ::= “typeId” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeId**
- a <*scoped_name*> that denotes the named IDL construct to which the repository identifier is assigned
- a string literal that must contain a valid repository identifier value

The <scoped_name> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface
- component
- home
- facet
- receptacle
- event sink

- event source
- finder
- factory
- value type
- value type member
- value box
- constant
- typedef
- exception
- attribute
- operation
- enum
- local

The value of the string literal is assigned as the repository identity of the specified type definition. This value will be returned as the RepositoryId by the interface repository definition object corresponding to the specified type definition. Language mappings constructs, such as Java helper classes, that return repository identifiers shall return the values declared for their corresponding definitions.

At most one repository identity declaration may occur for any named type definition. An attempt to re-define the repository identity for a type definition is illegal, regardless of the value of the re-definition.

If no explicit repository identity declaration exists for a type definition, the repository identifier for the type definition shall be an IDL format repository identifier, as defined in section 10.6.1 of the CORBA 2.3 specification.

14.1.7 Repository identifier prefix declaration

The syntax of a repository identifier prefix declaration is as follows:

<type_prefix_dcl> ::= “typePrefix” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeId**
- a <*scoped_name*> that denotes an IDL name scope to which the prefix applies
- a string literal that must contains the string to be pre-fixed to repository identifiers in the specified name scope

The <scoped_name> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface (including abstract interface)
- value type (including abstract, custom, and box value types)

- local interface
- specification scope (**::**)

The specified string is pre-fixed to the body of all repository identifiers in the specified name scope, whose values are assigned by default. To elaborate:

By “prefixed to the body of a repository identifier”, we mean that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon (“IDL:”) at the beginning of the identifier. A forward slash (‘/’) character is inserted between the end of the specified string and the remaining body of the repository identifier.

The prefix is only applied to repository identifiers whose values are not explicitly assigned by a typeId declaration. The prefix is applied to all such repository identifiers in the specified name scope, including the identifier of the construct that constitutes the name scope.

14.1.8 IDL Grammar modifications

In addition the extensions to IDL grammar specified in the previous sections, the following productions shall be modified to define the scopes in which local, typeId, and typePrefix may occur:

<definition> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <interface> “;”
| <module> “;”
| <value> “;”
| <local> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”

<export> ::= <type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
|<op_dcl> “ ;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”

14.1.9 Keywords

This specification defines the following new keywords in IDL:

import local localBase typeId typePrefix

14.1.10 Changes to the Attribute declaration syntax

The CORBA Component specification modifies the existing definition of attributes to add the ability to raise independent exceptions on the attribute's accessor and mutator operations. The modified syntax for attributes is as follows:

<attr_dcl> ::= <readonly_attr_spec>
| <attr_spec>

<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>
<readonly_attr_declarator>

<readonly_attr_declarator> ::= <simple_declarator> [<raises_expr>]
| <simple_declarator> { "," <simple_declarator> }*

<attr_dcl> ::= ["readonly"] "attribute" <param_type_spec>
<simple_declarator> { "," <simple_declarator> }*

<attr_spec> ::= "attribute" <param_type_spec> <attr_declarator>

<attr_declarator> ::= <simple_declarator> <attr_raises_expr>
| <simple_declarator> { "," <simple_declarator> }*

<attr_raises_expr> ::= <get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>

<get_excep_expr> ::= "getRaises" <exception_list>

<set_excep_expr> ::= "setRaises" <exception_list>

<exception_list> ::= "(" <scoped_name> { "," <scoped_name> } * ")"

These modifications to the existing attribute declaration syntax allow attribute get and set methods to raise user-defined exceptions. Note the following characteristics of the extended attribute declaration syntax:

- All existing attribute declarations using the previous syntax are still valid, and produce exactly the same results.
- When an attribute declaration raises an exception (on get, set or both), the declaration may not contain multiple declarators.

14.2 Changes to Object Services

14.2.1 Life Cycle Service

To support the factory design pattern for creating a component instance and to allow the server, rather than a client, to select from a group of functionally equivalent factories based on load or other server-side visible criteria, the following operation is added to the **FactoryFinder** interface of the **CosLifeCycle** module:

```
module CosLifeCycle {
        interface FactoryFinder {
        Factory find_factory (in Key factory_key) raises (noFactory);
        };
};
```

The parameters of the above operation are as defined by the LifeCycle service.

14.2.2 Transaction Service

No changes identified.

14.2.3 Security Service

No changes identified.

14.2.4 Name Service

No changes identified.

14.2.5 Notification Service

No changes identified.

Conformance Criteria 15

This chapter identifies the conformance points required for compliant implementations of the CORBA Component architecture.

15.1 Conformance Points

There are three conformance points that constitute CORBA Components

1. Changes to CORBA Core and COS Services.

   Chapter 14 identifies changes to CORBA Core and COS Services; compliant implementations of each shall implement these changes. Note that a CORBA ORB vendor need not provide implementations of Components aside from the changes made to the Core to support components.

   In Chapter 10, the MOF metamodel of the Interface Repository and the XMI format for the exchange of Interface Repository metadata is defined. The IDL for a MOF-compliant Interface Repository is defined as well. Support for the generation and consumption of the XMI metadata and for the MOF-compliant IDL is optional.

2. CORBA Components

   A CORBA component vendor shall support the following:

   - IDL extensions to support the client and server side component model including CIDL
   - A container for hosting CORBA components.
   - An installer object for deploying components. The installer object shall be capable of receiving XML deployment descriptors and associated zip files in the format defined in Section 9.4 on 243.

   All CORBA components, containers, and installers must work in a CORBA environment defined by conformance point 1 above. This CORBA environment need not be provided by the Component vendor i.e. a container vendor does not have to be an ORB vendor, and vice-versa.

Chapter 10 defines the MOF metamodel of the Packaging and Deployment descriptors for CORBA components. The XMI format for the exchange of Packaging and Deployment metadata is defined along with the MOF-compliant IDL for a packaging and deployment repository. Support for the generation and consumption of the XMI metadata and for the MOF-compliant IDL is optional.

3. EJB Support

   Chapter 11 defines how an Enterprise Java Bean can be supported in a CORBA Component Container, how a CORBA Component may provide an EJB facade, and how an EJB may provide a CORBA Component facade. A CORBA Component vendor may support some, all or none of these features.

15.2 A Note on Tools

Component implementations are expected to be supported by tools. It is not possible to define conformance points for tools, since a particular tool may only support part of the component development and deployment life-cycle. Hence a suite of tools may be needed. The Component architecture contains a number of definitions that are relevant to tools, including zip files and XML formats, as well as IDL interfaces for customization and installation. Although it cannot be enforced, tools are expected to conform to the relevant areas with which they are dealing. For example, a tool that generates implementations for a particular platform is expected to generate XML according to the `<implementation>` clauses in the DTD defined in Chapter 9.

IDL Summary A

This appendix summarizes all the IDL defined for the CORBA component model. The Component model is assumed to be part of the CORBA_3 level of the CORBA specification. This is reflected in the module structure proposed by this specification. The complete structure is a recommendation to the OMG regarding the structuring of CORBA_3 definitions. Within this overall approach, the modules unique to CORBA components are summarized, as they are defined in the body of this specification.

A.1 Module Architecture

The component submitters suggest the following structure for CORBA_3:

```
module CORBA_3 {
    // outer namespace for all CORBA_3 changes //;
    module Core {
        // namespace for changes to the CORBA Core //;
    };
    module Components {
        // namespace for all changes introduced by CORBA components //;
        // all interfaces visible to both clients and servers defined here //;
        module Deployment {
            // interfaces used to deploy Components //;
        };
        module Server {
            // interfaces used by the Server are defined in this namespace //:
        };
        module Container {
            // interfaces used to implement Container on the POA //;
        };
    }:
};
```

All of the changes to CORBA Core are defined with the **Core** module. Those changes introduced by the CORBA component specification will be summarized within this namespace in Appendix A.2. The submitters recommend that other changes to the Core introduce by new adopted technology (e.g. messaging) or a Core RTF also be defined this way.

The **Components** module is a namespace that includes all the additions for CORBA components defined by this specification. It includes two embedded module definitions for those interfaces defined for use by the component implementor (**module Server**) and the container implementor (**module Container**). Those interfaces which can be used by either the client or the component implementation are defined within the **Components** module.

A.2 The Core Module

The **Core** module defines all the changes made to the CORBA core to support components.

```
module CORBA {
    interface Object { // PIDL
        ...
        Object get_component ( );
    };
};
```

A.3 The Components Module

The **Components** module defines all the interfaces used to access or implement a CORBA component. The **Components** module has the following structure:

```
module Components {
    // namespace for all changes introduced by CORBA components //;
    // all interfaces visible to both clients and servers defined here //;
    module Server;
    // interfaces used by the Server are defined in this namespace //:
    };
    module Deployment {
    // interfaces used to install components in a container //;
    };
    module Container;
    // interfaces used to implement Container on the POA //;
    };
};
```

A.3.1 Interfaces Defined Within the Components Module

The interfaces defined within the **Components** module are accessible by either component clients or component implementors. Those interfaces (described in Chapter 5) are defined by the following IDL:

```
module Components {

    interface ComponentDef;

    typedef string FeatureName;
    typedef sequence<FeaureName> NameList;

    valuetype Cookie {
        private sequence<octet> cookieValue;
    };

    exception InvalidName { };
    exception InvalidConnection { };
    exception ExceededConnectionLimit { };
    exception AlreadyConnected { };
    exception NoConnection { };
    exception CookieRequired { };
    exception DuplicateKeyValue { };
    exception UnknownKeyValue { };
    exception BadEventType {
        CORBA::RepositoryId expected_event_type
    };
    exception HomeNotFound { };

    interface Navigation {

        valuetype FacetDescription {
            public RepositoryID InterfaceID;
            public FeatureName Name;
        };

        valuetype Facet : FacetDescription {
            public Object Ref;
        };

        typedef sequence<Facet> Facets;

        typedef sequence<FacetDescription>
            FacetDescriptions;

        Object provide_facet( in FeatureName name )
            raises (InvalidName);

        FacetDescriptions describe_facets();

        Facets provide_all_facets();

        Facets provide_named_facets(in NameList names)
            raises (InvalidName);

        boolean same_component( in Object ref );
```

```
};

valuetype ConnectionDescription {
    public Cookie ck;
    public Object objref;
};

typedef sequence<ConnectionDescription> ConnectedDescriptions;

interface Receptacles {

    Cookie connect (
        in FeatureName name,
        in Object connection )
    raises (
        InvalidName,
        InvalidConnection,
        AlreadyConnected,
        ExceededConnectionLimit);

    void disconnect (
        in FeatureName name,
        in Cookie ck)
    raises (
        InvalidName,
        InvalidConnection,
        CookieRequired,
        NoConnection);

    ConnectionList get_connections (in FeatureName name)
    raises (InvalidName);
};

struct Property {
    PropertyName name;
    PropertyValue value;
};

typedef sequence<Property> EventData;

abstract valuetype EventBase {};

interface EventConsumerBase {
    void push_event(in EventBase evt) raises (BadEventType);
};

interface Events {
    EventConsumerBase
    get_consumer(in FeatureName sinkName)
        raises(InvalidName);
```

```
        Cookie subscribe(in FeatureName publisherName,
            in EventConsumerBase subscriber)
            raises (InvalidName);
        void unsubscribe(in FeatureName publisherName,
            in Cookie ck)
            raises(InvalidName, InvalidConnection);
        void connect_consumer(in FeatureName emitterName,
            in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected);
        EventConsumerBase
        disconnect_consumer(in FeatureName sourceName)
            raises(InvalidName, NoConnection);

    };

    interface HomeBase {
        ComponentDef get_component_def();
        void destroy_component ( in ComponentBase comp);
    };

    interface KeylessHomeBase {
        ComponentBase create_component();
    };

    interface HomeFinder {
        HomeBase find_home_by_component_type (
            in CORBA::RepositoryId comprepid)
            raises (HomeNotFound);
        HomeBase find_home_by_home_type (
            in CORBA::RepositoryId homerepid)
            raises (HomeNotFound);
        HomeBase find_home_by_name (
            in string home_name)
            raises (HomeNotFound);
    };

    interface Configurator {
        void configure(in ComponentBase comp)
        raises WrongComponentType;
    };

    valuetype ConfigValue {
        FeatureName name;
        any value;
    };

    typedef sequence<ConfigValue> ConfigValues;

    interface StandardConfigurator : Configurator {
        void set_configuration (in ConfigValues descr);
    };
```

```
    interface HomeConfiguration : HomeBase {
        void set_configurator (in Configurator cfg);
        void set_configuration_values(
            in StandardConfigurator::ConfigValues config);
        void complete_component_configuration(in boolean b);
        void disable_home_configuration();
    };

    interface ComponentBase
    : Navigation, Receptacles, Events {
        ComponentDef get_component_def ( );
        HomeBase get_home( );
        void configuration_complete( );
        void destroy();
    };

};
```

A.3.2 Interfaces Defined Within the Persistence Module

The **Persistence** Module is an embedded namespace within the **Components** module that defines those interfaces (described in Chapter 6) used to support persistence for CORBA components. It is defined by the following IDL:

```
module Persistence {

    struct Property {
        string name;
        any    value;
    };

    typedef sequence<Property> PropertyList;
    typedef sequence<octet> PersistentId;
    typedef sequence<PersistentId> PersistentIdList;
    typedef string PSSTypeId;

    exception DoesNotExist {};

    local StorageHomeBase {

        TypeId managed_storage_type_id();
        IncarnationBase incarnate(in PersistentID pid)
        raises (DoesNotExist);
        void remove(in IncarnationBase inc) raises (DoesNotExist);
        void remove_by_pid(in PersistentID pid) raises (DoesNotExist);
        void flush() raises (PersistentStoreError);
        void refresh() raises (PersistentStoreError);

    };

    local KeylessStorageHomeBase : StorageHomeBase {
        PersistentID create_pid();
    };

    local PersistentStoreBase {
        void open(in string name, in PropertylList params)
        raises(NoPermission);
        void close();
        StorageHomeBase provide_storage_home(in string home_name)
        raises(NotFound);
        void flush() raises(PersistentStoreError);
        void flush_by_pids(in PersistentStoreIdList pids)
        raises(PersistentStoreError);
        void refresh() raises(PersistentStoreError);
        void refresh_by_pids(in PersistentStoreIdList pids)
        raises(PersistentStoreError);

    };

    local GenericPersistentStore
    : PersistentStoreBase {
        StorageHomeBase provide_storage_home_by_type(
            in CORBA::RepositoryId type_id)
            raises (NotFound);
```

```
    };
};
```

A.3.3 Interfaces Defined Within the Deployment Module

The **Deployment** Module is an embedded namespace within the **Components** module that defines those interfaces (described in Chapter 9) used to deploy CORBA components. It is defined by the following IDL:

```
module Deployment {

    enum AssemblyState {INACTIVE, INSERVICE};
    exception UnknownImplId { };
    exception InvalidLocation { };
    exception InvalidAssembly { };

    interface Installation {
        boolean install(in string implGUID, in string cmpntloc)
            raises InvalidLocation;
        boolean replace(in string implGUID, in string cmpntloc)
            raises InvalidLocation;
        boolean remove(in string implGUID)
            raises UnknownImplId;
        string get_Implementation(in string implGUID)
            raises UnknownImplId;
        string get...(in string key); // TBD
    };

    interface AssemblyFactory {
        Cookie create(in string assemblyloc)
            raises InvalidLocation;
        Assembly lookup(in Cookie c)
            raises InvalidAssembly;
        boolean destroy(in Cookie c)
            raises InvalidAssembly;
    };
```

Issue – Specify cookies.

```
    interface Assembly {
        boolean build();
        boolean tear_down();
        AssemblyState get_state();
    };

    interface ServerActivator {
        ComponentServer create_component_server();
    };
```

```
    interface ComponentServer {
        Container create_container(...); // params TBD
    };

    interface Container {
        HomeBase install_home();
    };
};
```

A.3.4 Interfaces Defined Within the Server Module

The **Server** Module is an embedded namespace within the **Components** module that defines those interfaces used by the developer to implement a CORBA component. Those interfaces (described in Chapter 7) are defined by the following IDL:

```
module Server {

    enum BadComponentReferenceReason {
        NON_LOCAL_REFERENCE,
        NON_COMPONENT_REFERENCE,
        WRONG_CONTAINER,
        NOT_CREATED_WITH_AID,
        NOT_CREATED_WITH_PID
    };

    enum Status {
            ACTIVE,
            MARKED_ROLLBACK,
            PREPARED,
            COMMITTED,
            ROLLED_BACK,
            NO_TRANSACTION,
            PREPARING,
            COMMITTING,
            ROLLING_BACK
    };

    const StoreType USER=0;
    const StoreType PSS=1;

    union StoreId switch StoreType {
        case USER : ApplId aid;
        case PSS : Persistence::PersistentId pid;
    };

    struct SegmentDescr {
        long segment;
        StoreId sid;
    }:

    typedef sequence<SegmentDescr> SegmentList;
    typedef SecurityLevel2::Credentials Principal;
    typedef sequence<octet> ApplId;
    typedef CORBA::NVList Criteria;

    exception NotFound { };
    exception IllegalState { };
    exception PolicyMismatch { };
    exception NoTransaction { };
    exception InvalidCookie{ };
    exception InvalidCategory { };
    exception UnknownActualHome { };
    exception RemoteHomeNotSupported { };
    exception ChannelUnavailable { };
    exception InvalidSubscription { };
    exception DuplicateTarget { };
```

```
exception NoSuchTarget { };
exception BadComponentReference {
    BadComponentReferenceReason reason
};
exception InvalidName { };
exception AlreadyBound { };

local ComponentContext {
    CORBA::Object get_reference ()
        raises (IllegalState);
    HomeBase get_home();
    Transaction get_transaction();
    HomeRegistration get_home_registration ();
    Security get_security();
    Events get_events();
};

local BaseOrigin {
    void req_passivate ()
        raises (PolicyMismatch);
};

local LocalCookie {
    boolean same_as (in LocalCookie cookie);
};

local Transaction {
    void begin ();
    void commit ()
        raises (NoTransaction);
    void rollback ()
        raises (NoTransaction);
    void set_rollback_only ()
        raises (NoTransaction);
    boolean get_rollback_only()
        raises (NoTransaction);
    LocalCookie suspend ()
        raises (NoTransaction);
    void resume (in LocalCookie cookie)
        raises (InvalidCookie);
    Status get_status();
    void set_timeout (in long to);
};

local HomeRegistration {
    void register_home (in HomeBase home);
    void unregister_home (in HomeBase home);
};
```

```
local RemoteHomeRegistration : HomeRegistration {
    void register_remote_home (
        in HomeBase rhome,
        in HomeBase ahome)
        raises (UnknownActualHome, RemoteHomeNotSupported);
};

local Security {
    Principal get_caller_identity();
    boolean is_caller_in_role (in Principal role);
};

local EnterpriseComponent {
};

local TransientContext : ComponentContext {
    TransientOrigin get_transient_origin();
};

local TransientOrigin : Origin {
    CORBA::Object create_ref (
        in CORBA::RepositoryId repid)
};

local ServiceComponent : EnterpriseComponent {
    void set_transient_context (in TransientContext ctx);
};

local SessionComponent : ServiceComponent {
    void activate();
    void passivate();
    void remove ();
};

local Synchronization {
    void before_completion ();
    void after_completion (
        in boolean committed);
};
```

```
local PersistentContext : ComponentContext {
    ComponentId get_component_id ()
        raises (IllegalState);
    PersistentOrigin get_persistent_origin();
    Storage get_storage ();
};

local ComponentId {
    void add_segment (in long segment,
        in StoreId sid)
        raises (DuplicateSegment);
    void set_segment (in long segment)
        raises (NoSuchSegment);
    long get_segment ();
    StoreId get_store_id ();
    StoreId get_component_id ();
    StoreId get_store_id_for_segment (in long segment)
        raises (NoSuchSegment);
    SegmentList get_segment_list ();
};

local PersistentOrigin : BaseOrigin {
    ComponentId create_cid_from_aid (
        in ApplId aid);
    ComponentId create_cid_from_pid (
        in Persistence::PersistentId pid);
    HomeBase get_home_by_cid (
        in ComponentId cid);
    CORBA::Object create_ref_from_cid (
        in CORBA::RepositoryId repid,
        in ComponentId cid,
        in Criteria crit);
    ComponentId get_cid_from_ref (
        in CORBA::Object ref)
        raises (BadComponentReference);
    ApplId get_aid_from_cid (
        in ComponentId cid)
        raises (BadComponentReference);
    Persistence::PersitentId get_pid_from_cid (
        in ComponentId cid)
        raises (BadComponentReference);
};

local Storage {
    Persistence::StorageHomeBase get_storage_home (
        in Persistence::StorageHomeId homeid)
        raises (InvalidCategory,
            Persistence::HomeNotAvailable);
    PrimaryKey get_primary_key ()
        raises (InvalidCategory);
};
```

```
    local PersistentOrigin : Origin {
        ComponentId create_cid_from_aid (
            in ApplId aid);
        ComponentId create_cid_from_pid (
            in Persistence::PersistentId pid);
        HomeBase get_home_by_cid (
            in ComponentId cid);
        CORBA::Object create_ref_from_cid (
            in CORBA::RepositoryId repid,
            in ComponentId id,
            in Criteria crit);
        ComponentId get_cid_from_ref (
            in CORBA::Object ref)
            raises (BadComponentReference);
        ApplId get_aid_from_cid (
            in ComponentId cid)
            raises (BadComponentReference);
        Persistence::PersitentId get_pid_from_cid (
            in ComponentId cid)
            raises (BadComponentReference);
    };

    local PersistentComponent : EnterpriseComponent {
        void set_persistent_context (in PersistentContext ctx);
        void unset_persistent_context ();
        void activate ();
        void load ();
        void store ();
        void passivate ();
        void remove ();
    };
};
```

A.3.5 Interfaces Defined Within the Container Module

The **Container** Module is an embedded namespace within the **Components** module that defines those interfaces (described in Chapter 8) used by the container implementor for CORBA component. The **Container** module defines the specialized **ServantManager**s for the components specification. It is defined by the following IDL:

```
module Container {

    native ServantFactory;
    typedef short Category;
    const Category EMPTY=0;
    const Category SESSION=1;
    const Category PROCESS=2;
    const Category ENTITY=3;
    // values 4-16383 reserved for future use //;
    // values 16384-32767 reserved for vendor extensions //
    typedef sequence<octet> ServantFactoryId;
    typedef sequence<octet> ServantId;

    exception InvalidCategory { };
    exception NoServantAvailable { };

    interface ContainerFactory {
        Container create_container (
            in Category cat)
            raises (InvalidCategory);
    };

    interface TransientServantLocator : PortableServer::ServantLocator {
        PortableServant::Servant register_servant (
            in ServantId sid,
            in PortableServer::Servant svt);
        void unregister_servant (
            in PortableServer::Servant svt);
        PortableServer::Servant lookup_servant (in ServantId sid);
        ServantFactory register_servant_factory (
            in ServantFactoryId sfid,
            in ServantFactory factory);
        void unregister_servant_factory (in ServantFactoryId sfid);
        ServantFactory lookup_servant_factory (
            in ServantFactoryId sfid);
    };
```

```
    interface PersistentServantLocator :
        PortableServer::ServantLocator {
            PortableServer::ObjectId create_oid_from_cid (
                in ServantFactoryId sfid,
                in Server::ComponentId cid);
            Server::ComponentId get_cid_from_oid (
                in PortableServer::ObjectId);
            ServantFactory register_servant_factory (
                in ServantFactoryId sfid,
                in ServantFactory factory);
            void unregister_servant_factory (in ServantFactory Id sfid);
            ServantFactory lookup_servant_factory (
                in ServantFactoryId sfid);
            void register_servant (in PortableServer::Server svt);
    };
};
```

XML DTDs B

B.1 softpkg.dtd

```
<!-- DTD for softpkg. Used to describe CORBA Component
     implementations. The root element is <softpkg>.
     Elements are listed alphabetically.
  -->
<!-- Simple xml link attributes based on W3C WD-xlink-19980303.
     May change when XLL is finalized. -->
<!ENTITY % simple-link-attributes "
     xml:link      CDATA               #FIXED 'SIMPLE'
     href          CDATA               #REQUIRED
">

<!ELEMENT author
     ( name
     | company
     | webpage
     )* >

<!ELEMENT code
     ( ( codebase
       | fileinarchive
       | link
       )
       , entrypoint?
     ) >
<!ATTLIST code
     type CDATA #IMPLIED >
<!-- If file not available locally, then download via codebase link -->
<!ELEMENT codebase EMPTY >
<!ATTLIST codebase
     filename CDATA #IMPLIED
     %simple-link-attributes; >
```

```
<!ELEMENT compiler EMPTY >
<!ATTLIST compiler
     name    CDATA #REQUIRED
     version CDATA #IMPLIED >

<!ELEMENT company ( #PCDATA ) >

<!ELEMENT dependency
     ( softpkg
     | codebase
     | fileinarchive
     | localfile
     | name
     ) >
<!ATTLIST dependency
     type   CDATA              #IMPLIED
     action (assert | install) "assert">

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT descriptor
     ( link
     | fileinarchive
     ) >
<!ATTLIST descriptor
     type   CDATA  #IMPLIED>

<!ELEMENT entrypoint ( #PCDATA) >

<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
     class     CDATA     #REQUIRED
     origin    CDATA     #REQUIRED
     id        ID        #IMPLIED
     extra     CDATA     #IMPLIED
     html-form CDATA     #IMPLIED >

<!-- The "fileinarchive" element is used to specify a file in the
archive.
     If the file is in another archive then link
     is used to point to the archive in which the file may be found.
  -->
<!ELEMENT fileinarchive
     ( link? ) >
<!ATTLIST fileinarchive
     name CDATA #REQUIRED >

<!ELEMENT idl (link | fileinarchive | repository) >
<!ATTLIST idl
     id CDATA #REQUIRED>
```

```
<!ELEMENT implementation
     ( description
     | code
     | compiler
     | dependency
     | descriptor
     | extension
     | programminglanguage
     | humanlanguage
     | os
     | propertyfile
     | processor
     | runtime
     | threadsafety
     )* >
<!ATTLIST implementation
     id ID #IMPLIED >

<!ELEMENT humanlanguage EMPTY >
<!ATTLIST humanlanguage
     name CDATA #REQUIRED >

<!ELEMENT license ( #PCDATA ) >
<!ATTLIST license
     %simple-link-attributes; >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
     %simple-link-attributes; >

<!-- A file that should be available in the local environment -->
<!ELEMENT localfile EMPTY >
<!ATTLIST localfile
     name CDATA #REQUIRED >

<!ELEMENT name ( #PCDATA ) >

<!ELEMENT os EMPTY >
<!ATTLIST os
     name    CDATA #REQUIRED
     version CDATA #IMPLIED>

<!ELEMENT pkgtype ( #PCDATA ) >

<!ELEMENT processor EMPTY >

<!ATTLIST processor
     name CDATA #REQUIRED >

<!ELEMENT programminglanguage EMPTY>
<!ATTLIST programminglanguage
     name CDATA #REQUIRED
     version  CDATA #IMPLIED >
```

```
<!ELEMENT propertyfile
     ( fileinarchive
     | link) >

<!ELEMENT repository EMPTY >
<!ATTLIST repository
     type CDATA #IMPLIED
     %simple-link-attributes; >

<!ELEMENT resource
     ( localfile
     | codebase
     ) >

<!ATTLIST resource
     type CDATA #IMPLIED >

<!ELEMENT runtime EMPTY >
<!ATTLIST runtime
     name    CDATA #REQUIRED
     version CDATA #IMPLIED>

<!ELEMENT softpkg
     ( title
     | pkgtype
     | author
     | description?
     | license
     | idl
     | propertyfile
     | dependency
     | descriptor
     | implementation
     | extension
     )* >
<!ATTLIST softpkg
     name    ID    #REQUIRED
     version CDATA #IMPLIED >

<!ELEMENT threadsafety EMPTY >
<!ATTLIST threadsafety
     level (none|class|instance) #REQUIRED >

<!-- DEVNOTE: or should we specify level1,level2??? -->
<!ELEMENT title ( #PCDATA ) >

<!ELEMENT webpage ( #PCDATA ) >
<!ATTLIST webpage
     %simple-link-attributes; >
```

B.2 corbacomponent.dtd

```
<!-- DTD for CORBA Component Descriptor. The root element is
     <corbacomponent>. Elements are listed alphabetically.
  -->
<!ELEMENT client EMPTY >

<!ELEMENT componentfeatures
    ( inheritscomponent?
    , supportsinterface*
    , ports
    , extension*
    ) >
<!ATTLIST componentfeatures
    name CDATA #REQUIRED
    repid ID #REQUIRED >

<!ELEMENT componentkind
     ( service
     | session
     | process
     | entity
     | unclassified
     ) >

<!ELEMENT corbacomponent
     ( corbaversion
     , repositoryid
     , componentkind
     , transaction
     , security?
     , eventpolicy?
     , threading
     , configurationcomplete
     , extendedpoapolicy*
     , repository?
     , componentfeatures+
     , interface*
     , extension*
     ) >

<!ELEMENT configurationcomplete EMPTY >
<!ATTLIST configurationcomplete
     set ( true | false ) #REQUIRED >

<!ELEMENT consumes EMPTY>
<!ATTLIST consumes
     consumesname CDATA #REQUIRED
     eventtype CDATA #REQUIRED
     eventname CDATA #REQUIRED >

<!ELEMENT corbaversion (#PCDATA) >
```

```
<!ELEMENT emits EMPTY>
<!ATTLIST emits
     emitsname CDATA #REQUIRED
     eventtype CDATA #REQUIRED
     eventname CDATA #REQUIRED >

<!ELEMENT entity
     ( servant
     , persistence
     ) >

<!ELEMENT eventpolicy EMPTY>
<!ATTLIST eventpolicy
     emit ( normal | default | transaction ) #IMPLIED
     consume (normal | default | transaction ) #IMPLIED >

<!ELEMENT extendedpoapolicy EMPTY>
<!ATTLIST extendedpoapolicy
     name CDATA #REQUIRED
     value CDATA #REQUIRED >

<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
     class     CDATA     #REQUIRED
     origin    CDATA     #REQUIRED
     id        ID        #IMPLIED
     extra     CDATA     #IMPLIED
     html-form CDATA     #IMPLIED >

<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
    repid IDREF #REQUIRED>

<!ELEMENT inheritsinterface EMPTY>
<!ATTLIST inheritsinterface
    repid IDREF #REQUIRED>

<!ELEMENT ins EMPTY>
<!ATTLIST ins
     name CDATA #REQUIRED >

<!ELEMENT interface ( inheritsinterface* ) >
<!ATTLIST interface
    name CDATA #REQUIRED
    repid ID #REQUIRED >

<!ELEMENT objref EMPTY>
<!ATTLIST objref
     string CDATA #REQUIRED >

<!ELEMENT persistence ( persistentstoreinfo? )>
<!ATTLIST persistence
     responsibility ( container | component ) #REQUIRED
     usepss ( true | false ) #REQUIRED >
```

```
<!ELEMENT persistentstoreinfo EMPTY>
<!ATTLIST persistentstoreinfo
     implementation CDATA #REQUIRED
     datastorename CDATA #REQUIRED
     datastoreid CDATA #REQUIRED >

<!ELEMENT poapolicies EMPTY>
<!ATTLIST poapolicies
     thread (ORB_CTRL_MODEL | SINGLE_THREAD_SAFE ) #REQUIRED
     lifespan (TRANSIENT | PERSISTENT ) #REQUIRED
     iduniqueness (UNIQUE_ID | MULTIPLE_ID) #REQUIRED
     idassignment (USER_ID | SYSTEM_ID) #REQUIRED
     servantretention (RETAIN | NON_RETAIN) #REQUIRED
     requestprocessing (USE_ACTIVE_OBJECT_MAP_ONLY
                       |USE_DEFAULT_SERVANT
                       |USE_SERVANT_MANAGER) #REQUIRED
     implicitactivation (IMPLICIT_ACTIVATION
                         |NON_IMPLICIT_ACTIVATION) #REQUIRED >

<!ELEMENT ports
     ( uses
     | provides
     | emits
     | consumes
     )* >

<!ELEMENT process
     ( servant
     , persistence
     ) >

<!ELEMENT provides EMPTY>
<!ATTLIST provides
     providesname CDATA #REQUIRED
     repid IDREF #REQUIRED >

<!ELEMENT repository ( ins | objref ) >
<!ATTLIST repository
     type CDATA #IMPLIED >

<!ELEMENT repositoryid EMPTY >
<!ATTLIST repositoryid
     repid IDREF #IMPLIED >

<!ELEMENT security
     ( securitycredentialkind ) >

<!ELEMENT securitycredentialkind
     ( client
     | system
     | specified
     ) >
```

```
<!ELEMENT servant EMPTY >
<!ATTLIST servant
     lifetime (process|method|transaction) #REQUIRED >

<!ELEMENT service EMPTY >
<!ELEMENT session
     ( servant ) >

<!ELEMENT specified EMPTY >
<!ATTLIST specified
     userid CDATA #REQUIRED >

<!ELEMENT supportsinterface EMPTY>
<!ATTLIST supportsinterface
     repid IDREF #REQUIRED >

<!ELEMENT system EMPTY >

<!ELEMENT threading EMPTY>
<!ATTLIST threading
     policy ( serialize | multithread ) #REQUIRED >

<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
     use (not-supported|required|supports|requires-new|mandatory|never)
#REQUIRED >

<!ELEMENT unclassified
     ( poapolicies
     , persistentstoreinfo
     ) >

<!ELEMENT uses EMPTY>
<!ATTLIST uses
     usesname CDATA #REQUIRED
     repid IDREF #REQUIRED >
```

B.3 *properties.dtd*

```
<!-- DTD for CORBA Component property file. The root element
     is <properties>. Elements are listed alphabetically.
  -->
<!ELEMENT choice ( #PCDATA ) >
<!ELEMENT choices ( choice+ ) >
<!ELEMENT defaultvalue ( #PCDATA ) >
<!ELEMENT description ( #PCDATA ) >
<!ELEMENT value ( #PCDATA ) >

<!ELEMENT properties
     ( description?
     , ( simple
       | sequence
       | struct
       )*
     ) >

<!ELEMENT simple
     ( description?
     , value
     , choices?
     , defaultvalue?
     ) >
<!ATTLIST simple
     name CDATA #IMPLIED
     type ( boolean
          | char
          | double
          | float
          | short
          | long
          | objref
          | octet
          | short
          | string
          | ulong
          | ushort
          ) #REQUIRED >

<!ELEMENT sequence
     ( description?
     , ( simple*
       | struct*
       | sequence*
       )
     ) >
<!ATTLIST sequence
     name CDATA #IMPLIED
     type CDATA #REQUIRED >
```

```
<!ELEMENT struct
     ( description?
     , ( simple
       | sequence
       | struct
       )*
     ) >
<!ATTLIST struct
     name CDATA #IMPLIED
     type CDATA #REQUIRED >
```

B.4 *componentassembly.dtd*

```
<!-- DTD for Component Assembly Descriptor. The root element
     is <componentassembly>. Elements are listed
     alphabetically.
  -->
<!-- Simple xml link attributes based on W3C WD-xlink-19980303.
     May change slightly when XLL is finalized.
  -->
<!ENTITY % simple-link-attributes "
     xml:link      CDATA              #FIXED 'SIMPLE'
     href          CDATA              #REQUIRED " >

<!-- If file not available locally, then download via codebase link -->
<!ELEMENT codebase EMPTY >
<!ATTLIST codebase
     filename CDATA #IMPLIED
     %simple-link-attributes; >

<!ELEMENT componentassembly
     ( componentfiles
     | partitioning
     | connections
     | extension
     )* >
<!ATTLIST componentassembly
     id ID #IMPLIED >

<!ELEMENT componentfile
     ( fileinarchive
     | codebase
     | link
     ) >
<!ATTLIST componentfile
     id ID #REQUIRED >

<!ELEMENT componentfileref EMPTY >
<!ATTLIST componentfileref
     idref IDREF #REQUIRED >

<!ELEMENT componentfiles
     ( componentfile+
     ) >

<!ELEMENT componentimplref EMPTY >
<!ATTLIST componentimplref
     idref CDATA #REQUIRED >
```

```
<!ELEMENT componentplacement
     ( usagename?
     , componentfileref
     , componentimplref?
     , propertiesfile?
     , stringifiedobjectref?
     , registerwithnaming*
     , registerwithtrader*
     , extension*
     ) >
<!ATTLIST componentplacement
     id          ID    #REQUIRED
     cardinality CDATA "1" >

<!ELEMENT connectevent
     ( emitingcomponent
     , consumingcomponent ) >
<!ATTLIST connectevent
     id ID #IMPLIED >

<!ELEMENT connectinterface
     ( usingcomponent
     , providingcomponent ) >
<!ATTLIST connectinterface
     id ID #IMPLIED >

<!ELEMENT connections
     ( connectinterface
     | connectevent
     | extension
     )* >

<!ELEMENT consumesidentifier ( #PCDATA ) >

<!ELEMENT consumingcomponent
     ( consumesidentifier
     , findby* )>
<!ATTLIST consumingcomponent
     idref IDREF #REQUIRED >

<!ELEMENT emitingcomponent
     ( emitsidentifier
     , findby* )>
<!ATTLIST emitingcomponent
     idref IDREF #REQUIRED >

<!ELEMENT emitsidentifier ( #PCDATA ) >
```

```
<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
     class     CDATA     #REQUIRED
     origin    CDATA     #REQUIRED
     id        ID        #IMPLIED
     extra     CDATA     #IMPLIED
     html-form CDATA     #IMPLIED >
<!-- The "fileinarchive" element is used to specify a file in the
archive.
     If the description file is independent of an archive then url
     is used to point to the archive in which the file may be found.
  -->
<!ELEMENT fileinarchive
     ( url? ) >
<!ATTLIST fileinarchive
     name CDATA #REQUIRED>

<!ELEMENT findby
     ( namingservice
     | stringifiedobjectref
     | installprocess
     | traderquery
     | extension
     ) >

<!ELEMENT hostcollocation
     ( usagename?
     , impltype?
     , ( componentplacement
       | processcollocation
       | extension
       )+
     ) >
<!ATTLIST hostcollocation
     id          ID    #IMPLIED
     cardinality CDATA "1" >

<!ELEMENT impltype EMPTY >
<!ATTLIST impltype
     language CDATA #REQUIRED
     version  CDATA #IMPLIED >

<!ELEMENT installprocess EMPTY >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
     %simple-link-attributes; >

<!ELEMENT namingservice EMPTY >

<!ELEMENT partitioning
     ( componentplacement
     | processcollocation
     | hostcollocation
```

```
     | extension
     )* >

<!ELEMENT processcollocation
     ( usagename?
     , impltype?
     , ( componentplacement
       | extension
       )+
     ) >
<!ATTLIST processcollocation
     id          ID    #IMPLIED
     cardinality CDATA "1" >

<!ELEMENT propertiesfile
     ( fileinarchive
     | codebase
     ) >

<!ELEMENT providesidentifier ( #PCDATA ) >

<!ELEMENT providingcomponent
     ( providesidentifier
     , findby* )>
<!ATTLIST providingcomponent
     idref IDREF #REQUIRED >

<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming
     name CDATA #REQUIRED >

<!ELEMENT registerwithtrader
     ( traderproperties ) >
<!ATTLIST registerwithtrader
     tradername CDATA #IMPLIED >

<!ELEMENT stringifiedobjectref ( #PCDATA ) >

<!ELEMENT traderconstraint ( #PCDATA ) >

<!ELEMENT traderexport
     ( traderservicetypename
     , traderproperties
     ) >

<!ELEMENT traderpolicy
     ( traderpolicyname
     , traderpolicyvalue
     ) >

<!ELEMENT traderpolicyname ( #PCDATA ) >

<!ELEMENT traderpolicyvalue ANY >

<!ELEMENT traderpreference ( #PCDATA ) >
```

```
<!ELEMENT traderproperties
     ( traderproperty+ ) >

<!ELEMENT traderproperty
     ( traderpropertyname
     , traderpropertyvalue
     ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ANY >

<!ELEMENT traderquery
     ( traderservicetypename
     , traderconstraint
     , traderpreference?
     , traderpolicy*
     , traderspecifiedprop*
     ) >

<!ELEMENT traderservicetypename ( #PCDATA ) >

<!ELEMENT traderspecifiedprop ( #PCDATA ) >

<!ELEMENT usagename ( #PCDATA ) >


<!ELEMENT usesidentifier ( #PCDATA ) >
<!ELEMENT usingcomponent
     ( usesidentifier
     , findby* )>
<!ATTLIST usingcomponent
     idref IDREF #REQUIRED >
```

MOF DTDs and IDL C

The XMI DTDs and IDL for the IR metamodel are presented in this appendix. The DTD for the Packaging and Deployment metamodel is also included. The DTDs are generated from the respective MOF-compliant metamodels described in Chapter 10 via the MOF-XML mapping contained in the OMG XMI specification. The IDL is derived from the IR metamodel via the MOF-IDL mapping contained in the OMG Meta Object Facility.

The IDL requires the inclusion of the reflective interfaces defined in ad/97-10-03 as part of the MOF specification. It was validated with the Visibroker 3.2 and OrbixWeb 3.0 IDL compilers.

C.1 IR Metamodel

C.1.1 XMI DTD

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI is the top-level XML element for XMI transfer text       -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI (XMI.header, XMI.content?, XMI.difference*,
               XMI.extensions*) >
<!ATTLIST XMI
           xmi.version CDATA #FIXED "1.0"
           timestamp CDATA #IMPLIED
           verified (true | false) #IMPLIED
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.header contains documentation and identifies the model,  -->
<!-- metamodel, and metametamodel                                 -->
<!-- ____________________________________________________________ -->
```

```
<!ELEMENT XMI.header (XMI.documentation?, XMI.model*, XMI.metamodel*,
                      XMI.metametamodel*) >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- documentation for transfer data                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.documentation (#PCDATA | XMI.owner | XMI.contact |
                             XMI.longDescription | XMI.shortDescription |
                             XMI.exporter | XMI.exporterVersion |
                             XMI.notice)* >

<!ELEMENT XMI.owner ANY >

<!ELEMENT XMI.contact ANY >

<!ELEMENT XMI.longDescription ANY >

<!ELEMENT XMI.shortDescription ANY >

<!ELEMENT XMI.exporter ANY >

<!ELEMENT XMI.exporterVersion ANY >

<!ELEMENT XMI.exporterID ANY >

<!ELEMENT XMI.notice ANY >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.element.att defines the attributes that each XML element -->
<!-- that corresponds to a metamodel class must have to conform to -->
<!-- the XMI specification.                                       -->
<!-- ____________________________________________________________ -->

<!ENTITY % XMI.element.att
               'xmi.id ID #IMPLIED xmi.label CDATA #IMPLIED xmi.uuid
                CDATA #IMPLIED ' >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.link.att defines the attributes that each XML element that -->
<!-- corresponds to a metamodel class must have to enable it to   -->
<!-- function as a simple XLink as well as refer to model         -->
<!-- constructs within the same XMI file.                         -->
<!-- ____________________________________________________________ -->

<!ENTITY % XMI.link.att
               'xml:link CDATA #IMPLIED inline (true | false) #IMPLIED
                actuate (show | user) #IMPLIED href CDATA #IMPLIED role
                CDATA #IMPLIED title CDATA #IMPLIED show (embed | replace
                | new) #IMPLIED behavior CDATA #IMPLIED xmi.idref IDREF
                #IMPLIED xmi.uuidref CDATA #IMPLIED' >
```

```
<!-- ____________________________________________________________________ -->
<!--                                                                        -->
<!-- XMI.model identifies the model(s) being transferred                    -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.model ANY >
<!ATTLIST XMI.model
            %XMI.link.att;
            xmi.name     CDATA #REQUIRED
            xmi.version  CDATA #IMPLIED
>

<!-- ____________________________________________________________________ -->
<!--                                                                        -->
<!-- XMI.metamodel identifies the metamodel(s) for the transferred          -->
<!-- data                                                                   -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.metamodel ANY >
<!ATTLIST XMI.metamodel
            %XMI.link.att;
            xmi.name     CDATA #REQUIRED
            xmi.version  CDATA #IMPLIED
>

<!-- ____________________________________________________________________ -->
<!--                                                                        -->
<!-- XMI.metametamodel identifies the metametamodel(s) for the              -->
<!-- transferred data                                                       -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.metametamodel ANY >
<!ATTLIST XMI.metametamodel
            %XMI.link.att;
            xmi.name     CDATA #REQUIRED
            xmi.version  CDATA #IMPLIED
>

<!-- ____________________________________________________________________ -->
<!--                                                                        -->
<!-- XMI.content is the actual data being transferred                       -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.content ANY >

<!-- ____________________________________________________________________ -->
<!--                                                                        -->
<!-- XMI.extensions contains data to transfer that does not conform         -->
<!-- to the metamodel(s) in the header                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.extensions ANY >
<!ATTLIST XMI.extensions
            xmi.extender CDATA #REQUIRED
```

```
>

<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- extension contains information related to a specific model           -->
<!-- construct that is not defined in the metamodel(s) in the header      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.extension ANY >
<!ATTLIST XMI.extension
            %XMI.element.att;
            %XMI.link.att;
            xmi.extender   CDATA #REQUIRED
            xmi.extenderID CDATA #REQUIRED
>

<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- XMI.difference holds XML elements representing differences to a      -->
<!-- base model                                                           -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.difference (XMI.difference | XMI.delete | XMI.add |
                          XMI.replace)* >
<!ATTLIST XMI.difference
            %XMI.element.att;
            %XMI.link.att;
>

<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- XMI.delete represents a deletion from a base model                   -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.delete EMPTY >
<!ATTLIST XMI.delete
            %XMI.element.att;
            %XMI.link.att;
>

<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- XMI.add represents an addition to a base model                       -->
<!-- ____________________________________________________________________ -->

<!ELEMENT XMI.add ANY >
<!ATTLIST XMI.add
            %XMI.element.att;
            %XMI.link.att;
            xmi.position CDATA "-1"
>

<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- XMI.replace represents the replacement of a model construct          -->
```

```
<!-- with another model construct in a base model                    -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.replace ANY >
<!ATTLIST XMI.replace
            %XMI.element.att;
            %XMI.link.att;
            xmi.position CDATA "-1"
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.reference may be used to refer to data types not defined in -->
<!-- the metamodel                                                -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
            %XMI.link.att;
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- This section contains the declaration of XML elements        -->
<!-- representing data types                                      -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.TypeDefinitions ANY >

<!ELEMENT XMI.field ANY >

<!ELEMENT XMI.seqItem ANY >

<!ELEMENT XMI.octetStream (#PCDATA) >

<!ELEMENT XMI.unionDiscrim ANY >

<!ELEMENT XMI.enum EMPTY >
<!ATTLIST XMI.enum
            xmi.value CDATA #REQUIRED
>

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
            %XMI.link.att;
            xmi.type CDATA #IMPLIED
            xmi.name CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTypeCode (XMI.CorbaTcAlias | XMI.CorbaTcStruct |
                             XMI.CorbaTcSequence | XMI.CorbaTcArray |
                             XMI.CorbaTcEnum | XMI.CorbaTcUnion |
                             XMI.CorbaTcExcept | XMI.CorbaTcString |
                             XMI.CorbaTcWstring | XMI.CorbaTcShort |
                             XMI.CorbaTcLong | XMI.CorbaTcUshort |
```

```
                                XMI.CorbaTcUlong | XMI.CorbaTcFloat |
                                XMI.CorbaTcDouble | XMI.CorbaTcBoolean |
                                XMI.CorbaTcChar | XMI.CorbaTcWchar |
                                XMI.CorbaTcOctet | XMI.CorbaTcAny |
                                XMI.CorbaTcTypeCode | XMI.CorbaTcPrincipal |
                                XMI.CorbaTcNull | XMI.CorbaTcVoid |
                                XMI.CorbaTcLongLong |
                                XMI.CorbaTcLongDouble) >
<!ATTLIST XMI.CorbaTypeCode
            %XMI.element.att;
>

<!ELEMENT XMI.CorbaTcAlias (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcAlias
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcStruct (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcStruct
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcField (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcField
            xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcSequence (XMI.CorbaTypeCode |
                               XMI.CorbaRecursiveType) >
<!ATTLIST XMI.CorbaTcSequence
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaRecursiveType EMPTY >
<!ATTLIST XMI.CorbaRecursiveType
            xmi.offset CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcArray (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcArray
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcObjRef EMPTY >
<!ATTLIST XMI.CorbaTcObjRef
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcEnum (XMI.CorbaTcEnumLabel) >
<!ATTLIST XMI.CorbaTcEnum
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
```

```
>

<!ELEMENT XMI.CorbaTcEnumLabel EMPTY >
<!ATTLIST XMI.CorbaTcEnumLabel
            xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnionMbr (XMI.CorbaTypeCode, XMI.any) >
<!ATTLIST XMI.CorbaTcUnionMbr
            xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnion (XMI.CorbaTypeCode, XMI.CorbaTcUnionMbr*) >
<!ATTLIST XMI.CorbaTcUnion
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcExcept (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcExcept
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcString EMPTY >
<!ATTLIST XMI.CorbaTcString
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcWstring EMPTY >
<!ATTLIST XMI.CorbaTcWstring
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcFixed EMPTY >
<!ATTLIST XMI.CorbaTcFixed
            xmi.tcDigits CDATA #REQUIRED
            xmi.tcScale  CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcShort EMPTY >

<!ELEMENT XMI.CorbaTcLong EMPTY >

<!ELEMENT XMI.CorbaTcUshort EMPTY >

<!ELEMENT XMI.CorbaTcUlong EMPTY >

<!ELEMENT XMI.CorbaTcFloat EMPTY >

<!ELEMENT XMI.CorbaTcDouble EMPTY >

<!ELEMENT XMI.CorbaTcBoolean EMPTY >

<!ELEMENT XMI.CorbaTcChar EMPTY >
```

```
<!ELEMENT XMI.CorbaTcWchar EMPTY >

<!ELEMENT XMI.CorbaTcOctet EMPTY >

<!ELEMENT XMI.CorbaTcAny EMPTY >

<!ELEMENT XMI.CorbaTcTypeCode EMPTY >

<!ELEMENT XMI.CorbaTcPrincipal EMPTY >

<!ELEMENT XMI.CorbaTcNull EMPTY >

<!ELEMENT XMI.CorbaTcVoid EMPTY >

<!ELEMENT XMI.CorbaTcLongLong EMPTY >

<!ELEMENT XMI.CorbaTcLongDouble EMPTY >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL PACKAGE: BaseIDL                                           -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ENTITY % BaseIDL.ParameterMode
           ' XMI.value ( PARAM_IN| PARAM_OUT| PARAM_INOUT) #REQUIRED'>

<!ENTITY % BaseIDL.PrimitiveKind
           ' XMI.value ( PK_NULL| PK_VOID| PK_SHORT| PK_LONG| PK_USHORT|
PK_ULONG|PK_FLOAT| PK_DOUBLE| PK_BOOLEAN| PK_CHAR| PK_OCTET|PK_ANY|
PK_TYPECODE| PK_PRINCIPAL| PK_STRING| PK_OBJREF|PK_LONGLONG|
PK_ULONGLONG| PK_LONGDOUBLE| PK_WCHAR| PK_WSTRING) #REQUIRED'>

<!ENTITY % BaseIDL.long
           ' XMI.value ( :) #REQUIRED'>

<!ENTITY % BaseIDL.DefinitionKind
           ' XMI.value ( DK_NONE| DK_ALL|DK_ATTRIBUTE| DK_CONSTANT|
DK_EXCEPTION| DK_INTERFACE|DK_MODULE| DK_OPERATION| DK_TYPEDEF|DK_ALIAS|
DK_STRUCT| DK_UNION| DK_ENUM|DK_PRIMITIVE| DK_STRING| DK_SEQUENCE|
DK_ARRAY|DK_REPOSITORY|DK_WSTRING| DK_FIXED) #REQUIRED'>


<!-- *****    BaseIDL.Contains   *******  -->

<!ELEMENT BaseIDL.Container.contents ( BaseIDL.ConstantDef
  |BaseIDL.Contained
  |BaseIDL.ModuleDef
  |BaseIDL.Container
  |BaseIDL.TypedefDef
  |BaseIDL.InterfaceDef
  |BaseIDL.StructDef
  |BaseIDL.UnionDef
```

```
  |BaseIDL.EnumDef
  |BaseIDL.AliasDef
  |BaseIDL.ValueMemberDef
  |BaseIDL.ValueDef
  |BaseIDL.ValueBoxDef
  |BaseIDL.OperationDef
  |BaseIDL.ExceptionDef
  |BaseIDL.AttributeDef
  |ComponentIDL.ProvidesDef
  |ComponentIDL.FactoryDef
  |ComponentIDL.FinderDef
  |ComponentIDL.EmitsDef
  |ComponentIDL.ConsumesDef
  |ComponentIDL.PrimaryKeyDef
  |ComponentIDL.UsesDef
  |ComponentIDL.PublishesDef )* >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Typed                               -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.Typed.idlType (BaseIDL.IDLType
  |BaseIDL.TypedefDef
  |BaseIDL.InterfaceDef
  |BaseIDL.StructDef
  |BaseIDL.UnionDef
  |BaseIDL.EnumDef
  |BaseIDL.AliasDef
  |BaseIDL.StringDef
  |BaseIDL.WstringDef
  |BaseIDL.FixedDef
  |BaseIDL.SequenceDef
  |BaseIDL.ArrayDef
  |BaseIDL.PrimitiveDef
  |BaseIDL.ValueDef
  |BaseIDL.ValueBoxDef) >

<!ENTITY % BaseIDL.TypedAssociations '(BaseIDL.Typed.idlType )' >

<!ELEMENT BaseIDL.Typed ((XMI.extension* ,   %BaseIDL.TypedAssociations; )
)?>

<!ATTLIST BaseIDL.Typed %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ParameterDef                        -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ParameterDef.identifier (#PCDATA|XMI.reference)*>
```

```
<!ELEMENT BaseIDL.ParameterDef.direction EMPTY>
<!ATTLIST BaseIDL.ParameterDef.direction %BaseIDL.ParameterMode;>

<!ENTITY % BaseIDL.ParameterDefProperties
'(BaseIDL.ParameterDef.identifier
   ,BaseIDL.ParameterDef.direction  )' >

<!ENTITY % BaseIDL.ParameterDefAssociations
'(%BaseIDL.TypedAssociations;)' >

<!ELEMENT BaseIDL.ParameterDef ( %BaseIDL.ParameterDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ParameterDefAssociations; ) )?>

<!ATTLIST BaseIDL.ParameterDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Contained                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.Contained.identifier (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.Contained.repositoryId (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.Contained.version (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.Contained.definedIn (BaseIDL.ModuleDef
  |BaseIDL.Container
  |BaseIDL.InterfaceDef
  |BaseIDL.ValueDef)?>

<!ENTITY % BaseIDL.ContainedProperties '(BaseIDL.Contained.identifier
   ,BaseIDL.Contained.repositoryId
   ,BaseIDL.Contained.version  )' >

<!ENTITY % BaseIDL.ContainedAssociations '(BaseIDL.Contained.definedIn?)'
>

<!ELEMENT BaseIDL.Contained ( %BaseIDL.ContainedProperties;
       ,(XMI.extension* ,   %BaseIDL.ContainedAssociations; ) )?>

<!ATTLIST BaseIDL.Contained %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ConstantDef                         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ConstantDef.constValue (XMI.any)>

<!ENTITY % BaseIDL.ConstantDefProperties '(%BaseIDL.ContainedProperties;
```

```
   ,BaseIDL.ConstantDef.constValue  )' >

<!ENTITY % BaseIDL.ConstantDefAssociations
'(%BaseIDL.TypedAssociations;,%BaseIDL.ContainedAssociations;)' >

<!ELEMENT BaseIDL.ConstantDef ( %BaseIDL.ConstantDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ConstantDefAssociations; ) )?>

<!ATTLIST BaseIDL.ConstantDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.IDLType                             -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.IDLType (EMPTY )>

<!ATTLIST BaseIDL.IDLType %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Container                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % BaseIDL.ContainerProperties '(%BaseIDL.ContainedProperties;)'
>

<!ENTITY % BaseIDL.ContainerAssociations
'(%BaseIDL.ContainedAssociations;)' >

<!ENTITY % BaseIDL.ContainerCompositions '(BaseIDL.Container.contents*)'
>

<!ELEMENT BaseIDL.Container ( %BaseIDL.ContainerProperties;
       ,(XMI.extension* ,   %BaseIDL.ContainerAssociations; )
       ,  %BaseIDL.ContainerCompositions; )?>

<!ATTLIST BaseIDL.Container %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ModuleDef                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ModuleDef.prefix (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.ModuleDefProperties '(%BaseIDL.ContainerProperties;
   ,BaseIDL.ModuleDef.prefix  )' >
```

```
<!ENTITY % BaseIDL.ModuleDefAssociations
'(%BaseIDL.ContainerAssociations;)' >

<!ENTITY % BaseIDL.ModuleDefCompositions
'(%BaseIDL.ContainerCompositions;)' >

<!ELEMENT BaseIDL.ModuleDef ( %BaseIDL.ModuleDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ModuleDefAssociations; )
       ,  %BaseIDL.ModuleDefCompositions; )?>

<!ATTLIST BaseIDL.ModuleDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.TypedefDef                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % BaseIDL.TypedefDefProperties '(%BaseIDL.ContainedProperties;)'
>

<!ENTITY % BaseIDL.TypedefDefAssociations
'(%BaseIDL.ContainedAssociations;)' >

<!ELEMENT BaseIDL.TypedefDef ( %BaseIDL.TypedefDefProperties;
       ,(XMI.extension* ,   %BaseIDL.TypedefDefAssociations; ) )?>

<!ATTLIST BaseIDL.TypedefDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.InterfaceDef                        -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.InterfaceDef.isAbstract EMPTY>
<!ATTLIST BaseIDL.InterfaceDef.isAbstract
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT BaseIDL.InterfaceDef.base (BaseIDL.InterfaceDef)*>

<!ENTITY % BaseIDL.InterfaceDefProperties '(%BaseIDL.ContainerProperties;
   ,BaseIDL.InterfaceDef.isAbstract  )' >

<!ENTITY % BaseIDL.InterfaceDefAssociations
'(%BaseIDL.ContainerAssociations;
   ,BaseIDL.InterfaceDef.base*)' >

<!ENTITY % BaseIDL.InterfaceDefCompositions
'(%BaseIDL.ContainerCompositions;)' >

<!ELEMENT BaseIDL.InterfaceDef ( %BaseIDL.InterfaceDefProperties;
       ,(XMI.extension* ,   %BaseIDL.InterfaceDefAssociations; )
```

```
       ,  %BaseIDL.InterfaceDefCompositions; )?>

<!ATTLIST BaseIDL.InterfaceDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Field                               -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.Field.identifier (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.FieldProperties '(BaseIDL.Field.identifier  )' >

<!ENTITY % BaseIDL.FieldAssociations '(%BaseIDL.TypedAssociations;)' >

<!ELEMENT BaseIDL.Field ( %BaseIDL.FieldProperties;
       ,(XMI.extension* ,   %BaseIDL.FieldAssociations; ) )?>

<!ATTLIST BaseIDL.Field %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.StructDef                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.StructDef.members (BaseIDL.Field)>

<!ENTITY % BaseIDL.StructDefProperties '(%BaseIDL.TypedefDefProperties;
   ,BaseIDL.StructDef.members +)' >

<!ENTITY % BaseIDL.StructDefAssociations
'(%BaseIDL.TypedefDefAssociations;)' >

<!ELEMENT BaseIDL.StructDef ( %BaseIDL.StructDefProperties;
       ,(XMI.extension* ,   %BaseIDL.StructDefAssociations; ) )?>

<!ATTLIST BaseIDL.StructDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.UnionDef                            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.UnionDef.unionMembers (BaseIDL.UnionField)>

<!ELEMENT BaseIDL.UnionDef.discriminatorType (BaseIDL.IDLType
  |BaseIDL.TypedefDef
  |BaseIDL.InterfaceDef
  |BaseIDL.StructDef
```

```
  |BaseIDL.UnionDef
  |BaseIDL.EnumDef
  |BaseIDL.AliasDef
  |BaseIDL.StringDef
  |BaseIDL.WstringDef
  |BaseIDL.FixedDef
  |BaseIDL.SequenceDef
  |BaseIDL.ArrayDef
  |BaseIDL.PrimitiveDef
  |BaseIDL.ValueDef
  |BaseIDL.ValueBoxDef) >

<!ENTITY % BaseIDL.UnionDefProperties '(%BaseIDL.TypedefDefProperties;
   ,BaseIDL.UnionDef.unionMembers +)' >

<!ENTITY % BaseIDL.UnionDefAssociations
'(%BaseIDL.TypedefDefAssociations;
   ,BaseIDL.UnionDef.discriminatorType )' >

<!ELEMENT BaseIDL.UnionDef ( %BaseIDL.UnionDefProperties;
       ,(XMI.extension* ,   %BaseIDL.UnionDefAssociations; ) )?>

<!ATTLIST BaseIDL.UnionDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.UnionField                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.UnionField.identifier (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.UnionField.label (XMI.any)>

<!ENTITY % BaseIDL.UnionFieldProperties '(BaseIDL.UnionField.identifier
   ,BaseIDL.UnionField.label  )' >

<!ENTITY % BaseIDL.UnionFieldAssociations '(%BaseIDL.TypedAssociations;)'
>

<!ELEMENT BaseIDL.UnionField ( %BaseIDL.UnionFieldProperties;
       ,(XMI.extension* ,   %BaseIDL.UnionFieldAssociations; ) )?>

<!ATTLIST BaseIDL.UnionField %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.EnumDef                             -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.EnumDef.members (#PCDATA|XMI.reference)*>
```

```
<!ENTITY % BaseIDL.EnumDefProperties '(%BaseIDL.TypedefDefProperties;
   ,BaseIDL.EnumDef.members +)' >

<!ENTITY % BaseIDL.EnumDefAssociations
'(%BaseIDL.TypedefDefAssociations;)' >

<!ELEMENT BaseIDL.EnumDef ( %BaseIDL.EnumDefProperties;
       ,(XMI.extension* ,   %BaseIDL.EnumDefAssociations; ) )?>

<!ATTLIST BaseIDL.EnumDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.AliasDef                            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % BaseIDL.AliasDefProperties '(%BaseIDL.TypedefDefProperties;)'
>

<!ENTITY % BaseIDL.AliasDefAssociations
'(%BaseIDL.TypedAssociations;,%BaseIDL.TypedefDefAssociations;)' >

<!ELEMENT BaseIDL.AliasDef ( %BaseIDL.AliasDefProperties;
       ,(XMI.extension* ,   %BaseIDL.AliasDefAssociations; ) )?>

<!ATTLIST BaseIDL.AliasDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.StringDef                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.StringDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.StringDefProperties '(BaseIDL.StringDef.bound  )' >

<!ELEMENT BaseIDL.StringDef ( %BaseIDL.StringDefProperties; )?>

<!ATTLIST BaseIDL.StringDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.WstringDef                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.WstringDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.WstringDefProperties '(BaseIDL.WstringDef.bound  )' >
```

```
<!ELEMENT BaseIDL.WstringDef ( %BaseIDL.WstringDefProperties; )?>

<!ATTLIST BaseIDL.WstringDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.FixedDef                            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.FixedDef.digits (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.FixedDef.scale (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.FixedDefProperties '(BaseIDL.FixedDef.digits
   ,BaseIDL.FixedDef.scale  )' >

<!ELEMENT BaseIDL.FixedDef ( %BaseIDL.FixedDefProperties; )?>

<!ATTLIST BaseIDL.FixedDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.SequenceDef                         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.SequenceDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.SequenceDefProperties '(BaseIDL.SequenceDef.bound  )' >

<!ENTITY % BaseIDL.SequenceDefAssociations
'(%BaseIDL.TypedAssociations;)' >

<!ELEMENT BaseIDL.SequenceDef ( %BaseIDL.SequenceDefProperties;
       ,(XMI.extension* ,   %BaseIDL.SequenceDefAssociations; ) )?>

<!ATTLIST BaseIDL.SequenceDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ArrayDef                            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ArrayDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL.ArrayDefProperties '(BaseIDL.ArrayDef.bound  )' >

<!ENTITY % BaseIDL.ArrayDefAssociations '(%BaseIDL.TypedAssociations;)' >

<!ELEMENT BaseIDL.ArrayDef ( %BaseIDL.ArrayDefProperties;
```

```
       ,(XMI.extension* ,   %BaseIDL.ArrayDefAssociations; ) )?>

<!ATTLIST BaseIDL.ArrayDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.PrimitiveDef                        -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.PrimitiveDef.kind EMPTY>
<!ATTLIST BaseIDL.PrimitiveDef.kind %BaseIDL.PrimitiveKind;>

<!ENTITY % BaseIDL.PrimitiveDefProperties '(BaseIDL.PrimitiveDef.kind  )'
>

<!ELEMENT BaseIDL.PrimitiveDef ( %BaseIDL.PrimitiveDefProperties; )?>

<!ATTLIST BaseIDL.PrimitiveDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ValueMemberDef                      -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ValueMemberDef.isPublicMember EMPTY>
<!ATTLIST BaseIDL.ValueMemberDef.isPublicMember
        XMI.value ( true | false ) #REQUIRED>

<!ENTITY % BaseIDL.ValueMemberDefProperties
'(%BaseIDL.ContainedProperties;
   ,BaseIDL.ValueMemberDef.isPublicMember  )' >

<!ENTITY % BaseIDL.ValueMemberDefAssociations
'(%BaseIDL.TypedAssociations;,%BaseIDL.ContainedAssociations;)' >

<!ELEMENT BaseIDL.ValueMemberDef ( %BaseIDL.ValueMemberDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ValueMemberDefAssociations; ) )?>

<!ATTLIST BaseIDL.ValueMemberDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ValueDef                            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT BaseIDL.ValueDef.isAbstract EMPTY>
<!ATTLIST BaseIDL.ValueDef.isAbstract
        XMI.value ( true | false ) #REQUIRED>
```

```
<!ELEMENT BaseIDL.ValueDef.isCustom EMPTY>
<!ATTLIST BaseIDL.ValueDef.isCustom
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT BaseIDL.ValueDef.isTruncatable EMPTY>
<!ATTLIST BaseIDL.ValueDef.isTruncatable
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT BaseIDL.ValueDef.interfaceDef (BaseIDL.InterfaceDef)?>

<!ELEMENT BaseIDL.ValueDef.base (BaseIDL.ValueDef)?>

<!ELEMENT BaseIDL.ValueDef.abstractBase (BaseIDL.ValueDef)*>

<!ENTITY % BaseIDL.ValueDefProperties '(%BaseIDL.ContainerProperties;
   ,BaseIDL.ValueDef.isAbstract
   ,BaseIDL.ValueDef.isCustom
   ,BaseIDL.ValueDef.isTruncatable  )' >

<!ENTITY % BaseIDL.ValueDefAssociations '(%BaseIDL.ContainerAssociations;
   ,BaseIDL.ValueDef.interfaceDef?
   ,BaseIDL.ValueDef.base?
   ,BaseIDL.ValueDef.abstractBase*)' >

<!ENTITY % BaseIDL.ValueDefCompositions
'(%BaseIDL.ContainerCompositions;)' >

<!ELEMENT BaseIDL.ValueDef ( %BaseIDL.ValueDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ValueDefAssociations; )
       ,  %BaseIDL.ValueDefCompositions; )?>

<!ATTLIST BaseIDL.ValueDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ValueBoxDef                         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % BaseIDL.ValueBoxDefProperties
'(%BaseIDL.TypedefDefProperties;)' >

<!ENTITY % BaseIDL.ValueBoxDefAssociations
'(%BaseIDL.TypedefDefAssociations;)' >

<!ELEMENT BaseIDL.ValueBoxDef ( %BaseIDL.ValueBoxDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ValueBoxDefAssociations; ) )?>

<!ATTLIST BaseIDL.ValueBoxDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.OperationDef                        -->
```

```
<!--                                                                    -->
<!-- ______________________________________________________________ -->

<!ELEMENT BaseIDL.OperationDef.isOneway EMPTY>
<!ATTLIST BaseIDL.OperationDef.isOneway
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT BaseIDL.OperationDef.parameters (BaseIDL.ParameterDef)>

<!ELEMENT BaseIDL.OperationDef.contexts (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL.OperationDef.exceptionDef (BaseIDL.ExceptionDef)*>

<!ENTITY % BaseIDL.OperationDefProperties '(%BaseIDL.ContainedProperties;
   ,BaseIDL.OperationDef.isOneway
   ,BaseIDL.OperationDef.parameters *
   ,BaseIDL.OperationDef.contexts *)' >

<!ENTITY % BaseIDL.OperationDefAssociations
'(%BaseIDL.TypedAssociations;,%BaseIDL.ContainedAssociations;
   ,BaseIDL.OperationDef.exceptionDef*)' >

<!ELEMENT BaseIDL.OperationDef ( %BaseIDL.OperationDefProperties;
       ,(XMI.extension* ,   %BaseIDL.OperationDefAssociations; ) )?>

<!ATTLIST BaseIDL.OperationDef %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: BaseIDL.ExceptionDef                          -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT BaseIDL.ExceptionDef.members (BaseIDL.Field)>

<!ENTITY % BaseIDL.ExceptionDefProperties '(%BaseIDL.ContainedProperties;
   ,BaseIDL.ExceptionDef.members *)' >

<!ENTITY % BaseIDL.ExceptionDefAssociations
'(%BaseIDL.ContainedAssociations;)' >

<!ELEMENT BaseIDL.ExceptionDef ( %BaseIDL.ExceptionDefProperties;
       ,(XMI.extension* ,   %BaseIDL.ExceptionDefAssociations; ) )?>

<!ATTLIST BaseIDL.ExceptionDef %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: BaseIDL.AttributeDef                          -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT BaseIDL.AttributeDef.isReadonly EMPTY>
```

```
<!ATTLIST BaseIDL.AttributeDef.isReadonly
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT BaseIDL.AttributeDef.setException (BaseIDL.ExceptionDef)*>

<!ELEMENT BaseIDL.AttributeDef.getException (BaseIDL.ExceptionDef)*>

<!ENTITY % BaseIDL.AttributeDefProperties '(%BaseIDL.ContainedProperties;
   ,BaseIDL.AttributeDef.isReadonly  )' >

<!ENTITY % BaseIDL.AttributeDefAssociations
'(%BaseIDL.TypedAssociations;,%BaseIDL.ContainedAssociations;
   ,BaseIDL.AttributeDef.setException*
   ,BaseIDL.AttributeDef.getException*)' >

<!ELEMENT BaseIDL.AttributeDef ( %BaseIDL.AttributeDefProperties;
       ,(XMI.extension* ,   %BaseIDL.AttributeDefAssociations; ) )?>

<!ATTLIST BaseIDL.AttributeDef %XMI.element.att; %XMI.link.att; >

<!ELEMENT BaseIDL ((BaseIDL.ParameterDef
  |BaseIDL.Typed
  |BaseIDL.ConstantDef
  |BaseIDL.Contained
  |BaseIDL.IDLType
  |BaseIDL.ModuleDef
  |BaseIDL.Container
  |BaseIDL.TypedefDef
  |BaseIDL.InterfaceDef
  |BaseIDL.Field
  |BaseIDL.StructDef
  |BaseIDL.UnionDef
  |BaseIDL.UnionField
  |BaseIDL.EnumDef
  |BaseIDL.AliasDef
  |BaseIDL.StringDef
  |BaseIDL.WstringDef
  |BaseIDL.FixedDef
  |BaseIDL.SequenceDef
  |BaseIDL.ArrayDef
  |BaseIDL.PrimitiveDef
  |BaseIDL.ValueMemberDef
  |BaseIDL.ValueDef
  |BaseIDL.ValueBoxDef
  |BaseIDL.OperationDef
  |BaseIDL.ExceptionDef
  |BaseIDL.AttributeDef)*)>
<!ATTLIST BaseIDL %XMI.element.att; %XMI.link.att;>


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE: ComponentIDL                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->
```

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: ComponentIDL.ComponentDef              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT ComponentIDL.ComponentDef.supports (BaseIDL.InterfaceDef
  |ComponentIDL.ComponentDef
  |ComponentIDL.HomeDef)?>

<!ENTITY % ComponentIDL.ComponentDefAssociations
'(%BaseIDL.InterfaceDefAssociations;
   ,ComponentIDL.ComponentDef.supports?)' >

<!ENTITY % ComponentIDL.ComponentDefCompositions
'(%BaseIDL.InterfaceDefCompositions;)' >

<!ELEMENT ComponentIDL.ComponentDef ((XMI.extension* ,
%ComponentIDL.ComponentDefAssociations; )
       ,  %ComponentIDL.ComponentDefCompositions; )?>

<!ATTLIST ComponentIDL.ComponentDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: ComponentIDL.ProvidesDef               -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT ComponentIDL.ProvidesDef.interface (BaseIDL.InterfaceDef
  |ComponentIDL.ComponentDef
  |ComponentIDL.HomeDef) >

<!ENTITY % ComponentIDL.ProvidesDefProperties
'(%BaseIDL.ContainedProperties;)' >

<!ENTITY % ComponentIDL.ProvidesDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.ProvidesDef.interface )' >

<!ELEMENT ComponentIDL.ProvidesDef ( %ComponentIDL.ProvidesDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.ProvidesDefAssociations; ) )?>

<!ATTLIST ComponentIDL.ProvidesDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: ComponentIDL.HomeDef                   -->
<!--                                                              -->
<!-- ____________________________________________________________ -->
```

```
<!ELEMENT ComponentIDL.HomeDef.manages (ComponentIDL.ComponentDef) >

<!ENTITY % ComponentIDL.HomeDefAssociations
'(%BaseIDL.InterfaceDefAssociations;
   ,ComponentIDL.HomeDef.manages )' >

<!ENTITY % ComponentIDL.HomeDefCompositions
'(%BaseIDL.InterfaceDefCompositions;)' >

<!ELEMENT ComponentIDL.HomeDef ((XMI.extension* ,
%ComponentIDL.HomeDefAssociations; )
       ,  %ComponentIDL.HomeDefCompositions; )?>

<!ATTLIST ComponentIDL.HomeDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: ComponentIDL.FactoryDef                -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ENTITY % ComponentIDL.FactoryDefProperties
'(%BaseIDL.OperationDefProperties;)' >

<!ENTITY % ComponentIDL.FactoryDefAssociations
'(%BaseIDL.OperationDefAssociations;)' >

<!ELEMENT ComponentIDL.FactoryDef ( %ComponentIDL.FactoryDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.FactoryDefAssociations; ) )?>

<!ATTLIST ComponentIDL.FactoryDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: ComponentIDL.FinderDef                 -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ENTITY % ComponentIDL.FinderDefProperties
'(%BaseIDL.OperationDefProperties;)' >

<!ENTITY % ComponentIDL.FinderDefAssociations
'(%BaseIDL.OperationDefAssociations;)' >

<!ELEMENT ComponentIDL.FinderDef ( %ComponentIDL.FinderDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.FinderDefAssociations; ) )?>

<!ATTLIST ComponentIDL.FinderDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: ComponentIDL.EmitsDef                  -->
```

```
<!--                                                                    -->
<!-- ______________________________________________________________ -->

<!ELEMENT ComponentIDL.EmitsDef.event (BaseIDL.ValueDef) >

<!ENTITY % ComponentIDL.EmitsDefProperties
'(%BaseIDL.ContainedProperties;)' >

<!ENTITY % ComponentIDL.EmitsDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.EmitsDef.event )' >

<!ELEMENT ComponentIDL.EmitsDef ( %ComponentIDL.EmitsDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.EmitsDefAssociations; ) )?>

<!ATTLIST ComponentIDL.EmitsDef %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: ComponentIDL.ConsumesDef              -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT ComponentIDL.ConsumesDef.event (BaseIDL.ValueDef) >

<!ENTITY % ComponentIDL.ConsumesDefProperties
'(%BaseIDL.ContainedProperties;)' >

<!ENTITY % ComponentIDL.ConsumesDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.ConsumesDef.event )' >

<!ELEMENT ComponentIDL.ConsumesDef ( %ComponentIDL.ConsumesDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.ConsumesDefAssociations; ) )?>

<!ATTLIST ComponentIDL.ConsumesDef %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: ComponentIDL.PrimaryKeyDef            -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT ComponentIDL.PrimaryKeyDef.type (BaseIDL.ValueDef) >

<!ENTITY % ComponentIDL.PrimaryKeyDefProperties
'(%BaseIDL.ContainedProperties;)' >

<!ENTITY % ComponentIDL.PrimaryKeyDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.PrimaryKeyDef.type )' >
```

```
<!ELEMENT ComponentIDL.PrimaryKeyDef (
%ComponentIDL.PrimaryKeyDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.PrimaryKeyDefAssociations; ) )?>

<!ATTLIST ComponentIDL.PrimaryKeyDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: ComponentIDL.UsesDef                    -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT ComponentIDL.UsesDef.multiple EMPTY>
<!ATTLIST ComponentIDL.UsesDef.multiple
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT ComponentIDL.UsesDef.interface (BaseIDL.InterfaceDef
  |ComponentIDL.ComponentDef
  |ComponentIDL.HomeDef) >

<!ENTITY % ComponentIDL.UsesDefProperties '(%BaseIDL.ContainedProperties;
   ,ComponentIDL.UsesDef.multiple  )' >

<!ENTITY % ComponentIDL.UsesDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.UsesDef.interface )' >

<!ELEMENT ComponentIDL.UsesDef ( %ComponentIDL.UsesDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.UsesDefAssociations; ) )?>

<!ATTLIST ComponentIDL.UsesDef %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: ComponentIDL.PublishesDef               -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT ComponentIDL.PublishesDef.event (BaseIDL.ValueDef) >

<!ENTITY % ComponentIDL.PublishesDefProperties
'(%BaseIDL.ContainedProperties;)' >

<!ENTITY % ComponentIDL.PublishesDefAssociations
'(%BaseIDL.ContainedAssociations;
   ,ComponentIDL.PublishesDef.event )' >

<!ELEMENT ComponentIDL.PublishesDef (
%ComponentIDL.PublishesDefProperties;
       ,(XMI.extension* ,   %ComponentIDL.PublishesDefAssociations; ) )?>

<!ATTLIST ComponentIDL.PublishesDef %XMI.element.att; %XMI.link.att; >
```

```
<!ELEMENT ComponentIDL ((ComponentIDL.ComponentDef
  |ComponentIDL.ProvidesDef
  |ComponentIDL.HomeDef
  |ComponentIDL.FactoryDef
  |ComponentIDL.FinderDef
  |ComponentIDL.EmitsDef
  |ComponentIDL.ConsumesDef
  |ComponentIDL.PrimaryKeyDef
  |ComponentIDL.UsesDef
  |ComponentIDL.PublishesDef)*)>
<!ATTLIST ComponentIDL %XMI.element.att; %XMI.link.att;>
```

C.1.2 IDL for the IR Metamodel

```
#include "Reflective.idl"

module BaseIDL
{
  interface ParameterDefClass;
  interface ParameterDef;
  typedef sequence<ParameterDef> ParameterDefUList;
  interface ConstantDefClass;
  interface ConstantDef;
  typedef sequence<ConstantDef> ConstantDefUList;
  interface TypedClass;
  interface Typed;
  typedef sequence<Typed> TypedSet;
  typedef sequence<Typed> TypedUList;
  interface ModuleDefClass;
  interface ModuleDef;
  typedef sequence<ModuleDef> ModuleDefUList;
  interface TypedefDefClass;
  interface TypedefDef;
  typedef sequence<TypedefDef> TypedefDefUList;
  interface InterfaceDefClass;
  interface InterfaceDef;
  typedef sequence<InterfaceDef> InterfaceDefSet;
  typedef sequence<InterfaceDef> InterfaceDefUList;
  interface FieldClass;
  interface Field;
  typedef sequence<Field> FieldUList;
  interface StructDefClass;
  interface StructDef;
  typedef sequence<StructDef> StructDefUList;
  interface UnionDefClass;
  interface UnionDef;
  typedef sequence<UnionDef> UnionDefSet;
  typedef sequence<UnionDef> UnionDefUList;
  interface EnumDefClass;
  interface EnumDef;
  typedef sequence<EnumDef> EnumDefUList;
  interface AliasDefClass;
  interface AliasDef;
  typedef sequence<AliasDef> AliasDefUList;
  interface IDLTypeClass;
  interface IDLType;
  typedef sequence<IDLType> IDLTypeUList;
  interface StringDefClass;
  interface StringDef;
  typedef sequence<StringDef> StringDefUList;
  interface WstringDefClass;
  interface WstringDef;
```

```
    typedef sequence<WstringDef> WstringDefUList;
    interface FixedDefClass;
    interface FixedDef;
    typedef sequence<FixedDef> FixedDefUList;
    interface SequenceDefClass;
    interface SequenceDef;
    typedef sequence<SequenceDef> SequenceDefUList;
    interface ArrayDefClass;
    interface ArrayDef;
    typedef sequence<ArrayDef> ArrayDefUList;
    interface PrimitiveDefClass;
    interface PrimitiveDef;
    typedef sequence<PrimitiveDef> PrimitiveDefUList;
    interface UnionFieldClass;
    interface UnionField;
    typedef sequence<UnionField> UnionFieldUList;
    interface ContainerClass;
    interface Container;
    typedef sequence<Container> ContainerUList;
    interface ValueMemberDefClass;
    interface ValueMemberDef;
    typedef sequence<ValueMemberDef> ValueMemberDefUList;
    interface ValueDefClass;
    interface ValueDef;
    typedef sequence<ValueDef> ValueDefSet;
    typedef sequence<ValueDef> ValueDefUList;
    interface ValueBoxDefClass;
    interface ValueBoxDef;
    typedef sequence<ValueBoxDef> ValueBoxDefUList;
    interface OperationDefClass;
    interface OperationDef;
    typedef sequence<OperationDef> OperationDefSet;
    typedef sequence<OperationDef> OperationDefUList;
    interface ExceptionDefClass;
    interface ExceptionDef;
    typedef sequence<ExceptionDef> ExceptionDefSet;
    typedef sequence<ExceptionDef> ExceptionDefUList;
    interface ContainedClass;
    interface Contained;
    typedef sequence<Contained> ContainedSet;
    typedef sequence<Contained> ContainedUList;
    interface AttributeDefClass;
    interface AttributeDef;
    typedef sequence<AttributeDef> AttributeDefSet;
    typedef sequence<AttributeDef> AttributeDefUList;
    interface BaseIDLPackage;
    enum PrimitiveKind { PK_NULL, PK_VOID, PK_SHORT, PK_LONG,
PK_USHORT, PK_ULONG,PK_FLOAT, PK_DOUBLE, PK_BOOLEAN,
PK_CHAR, PK_OCTET,PK_ANY, PK_TYPECODE, PK_PRINCIPAL,
PK_STRING, PK_OBJREF,PK_LONGLONG, PK_ULONGLONG,
PK_LONGDOUBLE, PK_WCHAR, PK_WSTRING };
```

```
  enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
  enum DefinitionKind { DK_NONE, DK_ALL,DK_ATTRIBUTE,
DK_CONSTANT, DK_EXCEPTION, DK_INTERFACE,DK_MODULE,
DK_OPERATION, DK_TYPEDEF,DK_ALIAS, DK_STRUCT, DK_UNION,
DK_ENUM,DK_PRIMITIVE, DK_STRING, DK_SEQUENCE,
DK_ARRAY,DK_REPOSITORY,DK_WSTRING, DK_FIXED };
  typedef sequence<string> StringList;

  interface TypedClass : Reflective::RefObject
  {
    readonly attribute TypedUList all_of_kind_typed;
  };

  interface Typed : TypedClass
  {
    IDLType idl_type ()
      raises (Reflective::SemanticError);
    void set_idl_type (in IDLType new_value)
      raises (Reflective::SemanticError);
  }; // end of interface Typed

  interface ParameterDefClass : TypedClass
  {
    readonly attribute ParameterDefUList all_of_kind_parameter_def;
    readonly attribute ParameterDefUList all_of_type_parameter_def;
    ParameterDef create_parameter_def (
      in string identifier,
      in ParameterMode direction)
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface ParameterDef : ParameterDefClass, Typed
  {
    string identifier ()
      raises (Reflective::SemanticError);
    void set_identifier (in string new_value)
      raises (Reflective::SemanticError);
    ParameterMode direction ()
      raises (Reflective::SemanticError);
    void set_direction (in ParameterMode new_value)
      raises (Reflective::SemanticError);
  }; // end of interface ParameterDef

  interface ContainedClass : Reflective::RefObject
  {
    readonly attribute ContainedUList all_of_kind_contained;
  };

  interface Contained : ContainedClass
```

```
{
  string identifier ()
    raises (Reflective::SemanticError);
  void set_identifier (in string new_value)
    raises (Reflective::SemanticError);
  string repository_id ()
    raises (Reflective::SemanticError);
  void set_repository_id (in string new_value)
    raises (Reflective::SemanticError);
  string version ()
    raises (Reflective::SemanticError);
  void set_version (in string new_value)
    raises (Reflective::SemanticError);
  string absolute_name ()
    raises (Reflective::SemanticError);
  Container defined_in ()
    raises (
      Reflective::NotSet,
      Reflective::SemanticError);
  void set_defined_in (in Container new_value)
    raises (Reflective::SemanticError);
  void unset_defined_in ()
    raises (Reflective::SemanticError);
}; // end of interface Contained

interface ConstantDefClass : TypedClass, ContainedClass
{
  readonly attribute ConstantDefUList all_of_kind_constant_def;
  readonly attribute ConstantDefUList all_of_type_constant_def;
  ConstantDef create_constant_def (
    in string identifier,
    in string repository_id,
    in string version,
    in any const_value)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ConstantDef : ConstantDefClass, Typed, Contained
{
  any const_value ()
    raises (Reflective::SemanticError);
  void set_const_value (in any new_value)
    raises (Reflective::SemanticError);
}; // end of interface ConstantDef

interface ContainerClass : ContainedClass
{
  readonly attribute ContainerUList all_of_kind_container;
};
```

```
interface Container : ContainerClass, Contained
{
  ContainedSet lookup_name(
    inout boolean exclude_inherited,
    inout DefinitionKind limit_to_type,
    inout long levels_to_search,
    inout string search_name)
    raises (Reflective::SemanticError);
  Contained lookup(
    inout string search_name)
    raises (Reflective::SemanticError);
  ContainedSet get_filtered_contents(
    inout boolean include_inherited,
    inout DefinitionKind limit_to_type)
    raises (Reflective::SemanticError);
  ContainedSet contents ()
    raises (Reflective::SemanticError);
  void set_contents (in ContainedSet new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void unset_contents ()
    raises (Reflective::SemanticError);
  void add_contents (in Contained new_value)
    raises (Reflective::StructuralError);
  void modify_contents (
    in Contained old_value,
    in Contained new_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_contents (in Contained old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface Container

interface ModuleDefClass : ContainerClass
{
  readonly attribute ModuleDefUList all_of_kind_module_def;
  readonly attribute ModuleDefUList all_of_type_module_def;
  ModuleDef create_module_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string prefix)
    raises (
      Reflective::SemanticError,
```

```
        Reflective::ConstraintError);
};

interface ModuleDef : ModuleDefClass, Container
{
  string prefix ()
    raises (Reflective::SemanticError);
  void set_prefix (in string new_value)
    raises (Reflective::SemanticError);
}; // end of interface ModuleDef

interface IDLTypeClass : Reflective::RefObject
{
  readonly attribute IDLTypeUList all_of_kind_idltype;
};

interface IDLType : IDLTypeClass
{
  TypeCode type_code ()
    raises (Reflective::SemanticError);
  void set_type_code (in TypeCode new_value)
    raises (Reflective::SemanticError);
}; // end of interface IDLType

interface TypedefDefClass : IDLTypeClass, ContainedClass
{
  readonly attribute TypedefDefUList all_of_kind_typedef_def;
};

interface TypedefDef : TypedefDefClass, IDLType, Contained
{
}; // end of interface TypedefDef

interface InterfaceDefClass : IDLTypeClass, ContainerClass
{
  readonly attribute InterfaceDefUList all_of_kind_interface_def;
  readonly attribute InterfaceDefUList all_of_type_interface_def;
  InterfaceDef create_interface_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_abstract)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface InterfaceDef : InterfaceDefClass, IDLType, Container
{
  boolean is_abstract ()
    raises (Reflective::SemanticError);
```

```
    void set_is_abstract (in boolean new_value)
      raises (Reflective::SemanticError);
    InterfaceDefSet base ()
      raises (Reflective::SemanticError);
    void set_base (in InterfaceDefSet new_value)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void unset_base ()
      raises (Reflective::SemanticError);
    void add_base (in InterfaceDef new_value)
      raises (Reflective::StructuralError);
    void modify_base (
      in InterfaceDef old_value,
      in InterfaceDef new_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove_base (in InterfaceDef old_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface InterfaceDef

  interface FieldClass : TypedClass
  {
    readonly attribute FieldULList all_of_kind_field;
    readonly attribute FieldULList all_of_type_field;
    Field create_field (
      in string identifier)
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface Field : FieldClass, Typed
  {
    string identifier ()
      raises (Reflective::SemanticError);
    void set_identifier (in string new_value)
      raises (Reflective::SemanticError);
  }; // end of interface Field

  interface StructDefClass : TypedefDefClass
  {
    readonly attribute StructDefULList all_of_kind_struct_def;
    readonly attribute StructDefULList all_of_type_struct_def;
    StructDef create_struct_def (
      in string identifier,
```

```
    in string repository_id,
    in string version,
    in FieldULList members)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface StructDef : StructDefClass, TypedefDef
{
  FieldULList members ()
    raises (Reflective::SemanticError);
  void set_members (in FieldULList new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_members (in Field new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_members_before (
    in Field new_value,
    in Field before_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_members_at (
    in Field new_value,
    in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_members (
    in Field old_value,
    in Field new_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_members_at (
    in Field new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_members (in Field old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
```

```
  void remove_members_at (in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
}; // end of interface StructDef

interface UnionDefClass : TypedefDefClass
{
  readonly attribute UnionDefUList all_of_kind_union_def;
  readonly attribute UnionDefUList all_of_type_union_def;
  UnionDef create_union_def (
    in string identifier,
    in string repository_id,
    in string version,
    in UnionFieldUList union_members)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface UnionDef : UnionDefClass, TypedefDef
{
  UnionFieldUList union_members ()
    raises (Reflective::SemanticError);
  void set_union_members (in UnionFieldUList new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_union_members (in UnionField new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_union_members_before (
    in UnionField new_value,
    in UnionField before_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_union_members_at (
    in UnionField new_value,
    in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_union_members (
    in UnionField old_value,
    in UnionField new_value)
    raises (
```

```
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_union_members_at (
    in UnionField new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_union_members (in UnionField old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_union_members_at (in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
  IDLType discriminator_type ()
    raises (Reflective::SemanticError);
  void set_discriminator_type (in IDLType new_value)
    raises (Reflective::SemanticError);
}; // end of interface UnionDef

interface EnumDefClass : TypedefDefClass
{
  readonly attribute EnumDefUList all_of_kind_enum_def;
  readonly attribute EnumDefUList all_of_type_enum_def;
  EnumDef create_enum_def (
    in string identifier,
    in string repository_id,
    in string version,
    in StringList members)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface EnumDef : EnumDefClass, TypedefDef
{
  StringList members ()
    raises (Reflective::SemanticError);
  void set_members (in StringList new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_members (in string new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void add_members_before (
```

```
    in string new_value,
    in string before_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_members_at (
    in string new_value,
    in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_members (
    in string old_value,
    in string new_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_members_at (
    in string new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_members (in string old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_members_at (in unsigned long position)
    raises (
      Reflective::StructuralError,
      Reflective::BadPosition,
      Reflective::SemanticError);
}; // end of interface EnumDef

interface AliasDefClass : TypedefDefClass, TypedClass
{
  readonly attribute AliasDefUList all_of_kind_alias_def;
  readonly attribute AliasDefUList all_of_type_alias_def;
  AliasDef create_alias_def (
    in string identifier,
    in string repository_id,
    in string version)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface AliasDef : AliasDefClass, TypedefDef, Typed
```

```
{
}; // end of interface AliasDef

interface StringDefClass : IDLTypeClass
{
  readonly attribute StringDefUList all_of_kind_string_def;
  readonly attribute StringDefUList all_of_type_string_def;
  StringDef create_string_def (
    in unsigned long bound)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface StringDef : StringDefClass, IDLType
{
  unsigned long bound ()
    raises (Reflective::SemanticError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::SemanticError);
}; // end of interface StringDef

interface WstringDefClass : IDLTypeClass
{
  readonly attribute WstringDefUList all_of_kind_wstring_def;
  readonly attribute WstringDefUList all_of_type_wstring_def;
  WstringDef create_wstring_def (
    in unsigned long bound)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface WstringDef : WstringDefClass, IDLType
{
  unsigned long bound ()
    raises (Reflective::SemanticError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::SemanticError);
}; // end of interface WstringDef

interface FixedDefClass : IDLTypeClass
{
  readonly attribute FixedDefUList all_of_kind_fixed_def;
  readonly attribute FixedDefUList all_of_type_fixed_def;
  FixedDef create_fixed_def (
    in unsigned short digits,
    in short scale)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
```

```
};

interface FixedDef : FixedDefClass, IDLType
{
  unsigned short digits ()
    raises (Reflective::SemanticError);
  void set_digits (in unsigned short new_value)
    raises (Reflective::SemanticError);
  short scale ()
    raises (Reflective::SemanticError);
  void set_scale (in short new_value)
    raises (Reflective::SemanticError);
}; // end of interface FixedDef

interface SequenceDefClass : TypedClass, IDLTypeClass
{
  readonly attribute SequenceDefUList all_of_kind_sequence_def;
  readonly attribute SequenceDefUList all_of_type_sequence_def;
  SequenceDef create_sequence_def (
    in unsigned long bound)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface SequenceDef : SequenceDefClass, Typed, IDLType
{
  unsigned long bound ()
    raises (Reflective::SemanticError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::SemanticError);
}; // end of interface SequenceDef

interface ArrayDefClass : TypedClass, IDLTypeClass
{
  readonly attribute ArrayDefUList all_of_kind_array_def;
  readonly attribute ArrayDefUList all_of_type_array_def;
  ArrayDef create_array_def (
    in unsigned long bound)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ArrayDef : ArrayDefClass, Typed, IDLType
{
  unsigned long bound ()
    raises (Reflective::SemanticError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::SemanticError);
}; // end of interface ArrayDef
```

```
interface PrimitiveDefClass : IDLTypeClass
{
  readonly attribute PrimitiveDefUList all_of_kind_primitive_def;
  readonly attribute PrimitiveDefUList all_of_type_primitive_def;
  PrimitiveDef create_primitive_def (
    in PrimitiveKind kind)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface PrimitiveDef : PrimitiveDefClass, IDLType
{
  PrimitiveKind kind ()
    raises (Reflective::SemanticError);
  void set_kind (in PrimitiveKind new_value)
    raises (Reflective::SemanticError);
}; // end of interface PrimitiveDef

interface UnionFieldClass : TypedClass
{
  readonly attribute UnionFieldUList all_of_kind_union_field;
  readonly attribute UnionFieldUList all_of_type_union_field;
  UnionField create_union_field (
    in string identifier,
    in any label)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface UnionField : UnionFieldClass, Typed
{
  string identifier ()
    raises (Reflective::SemanticError);
  void set_identifier (in string new_value)
    raises (Reflective::SemanticError);
  any label ()
    raises (Reflective::SemanticError);
  void set_label (in any new_value)
    raises (Reflective::SemanticError);
}; // end of interface UnionField

interface ValueMemberDefClass : TypedClass, ContainedClass
{
  readonly attribute ValueMemberDefUList all_of_kind_value_member_def;
  readonly attribute ValueMemberDefUList all_of_type_value_member_def;
  ValueMemberDef create_value_member_def (
    in string identifier,
    in string repository_id,
```

```
    in string version,
    in boolean is_public_member)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ValueMemberDef : ValueMemberDefClass, Typed, Contained
{
  boolean is_public_member ()
    raises (Reflective::SemanticError);
  void set_is_public_member (in boolean new_value)
    raises (Reflective::SemanticError);
}; // end of interface ValueMemberDef

interface ValueDefClass : ContainerClass, IDLTypeClass
{
  readonly attribute ValueDefUList all_of_kind_value_def;
  readonly attribute ValueDefUList all_of_type_value_def;
  ValueDef create_value_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_abstract,
    in boolean is_custom,
    in boolean is_truncatable)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ValueDef : ValueDefClass, Container, IDLType
{
  boolean is_abstract ()
    raises (Reflective::SemanticError);
  void set_is_abstract (in boolean new_value)
    raises (Reflective::SemanticError);
  boolean is_custom ()
    raises (Reflective::SemanticError);
  void set_is_custom (in boolean new_value)
    raises (Reflective::SemanticError);
  boolean is_truncatable ()
    raises (Reflective::SemanticError);
  void set_is_truncatable (in boolean new_value)
    raises (Reflective::SemanticError);
  InterfaceDef interface_def ()
    raises (
      Reflective::NotSet,
      Reflective::SemanticError);
  void set_interface_def (in InterfaceDef new_value)
```

```
      raises (Reflective::SemanticError);
    void unset_interface_def ()
      raises (Reflective::SemanticError);
    ValueDef base ()
      raises (
        Reflective::NotSet,
        Reflective::SemanticError);
    void set_base (in ValueDef new_value)
      raises (Reflective::SemanticError);
    void unset_base ()
      raises (Reflective::SemanticError);
    ValueDefSet abstract_base ()
      raises (Reflective::SemanticError);
    void set_abstract_base (in ValueDefSet new_value)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void unset_abstract_base ()
      raises (Reflective::SemanticError);
    void add_abstract_base (in ValueDef new_value)
      raises (Reflective::StructuralError);
    void modify_abstract_base (
      in ValueDef old_value,
      in ValueDef new_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove_abstract_base (in ValueDef old_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface ValueDef

  interface ValueBoxDefClass : TypedefDefClass
  {
    readonly attribute ValueBoxDefUList all_of_kind_value_box_def;
    readonly attribute ValueBoxDefUList all_of_type_value_box_def;
    ValueBoxDef create_value_box_def (
      in string identifier,
      in string repository_id,
      in string version)
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface ValueBoxDef : ValueBoxDefClass, TypedefDef
  {
  }; // end of interface ValueBoxDef
```

```
interface OperationDefClass : TypedClass, ContainedClass
{
  readonly attribute OperationDefUList all_of_kind_operation_def;
  readonly attribute OperationDefUList all_of_type_operation_def;
  OperationDef create_operation_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_oneway,
    in ParameterDefUList parameters,
    in StringList contexts)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface OperationDef : OperationDefClass, Typed, Contained
{
  boolean is_oneway ()
    raises (Reflective::SemanticError);
  void set_is_oneway (in boolean new_value)
    raises (Reflective::SemanticError);
  ParameterDefUList parameters ()
    raises (Reflective::SemanticError);
  void set_parameters (in ParameterDefUList new_value)
    raises (Reflective::SemanticError);
  void unset_parameters ()
    raises (Reflective::SemanticError);
  void add_parameters (in ParameterDef new_value)
    raises (Reflective::SemanticError);
  void add_parameters_before (
    in ParameterDef new_value,
    in ParameterDef before_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_parameters_at (
    in ParameterDef new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_parameters (
    in ParameterDef old_value,
    in ParameterDef new_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_parameters_at (
    in ParameterDef new_value,
```

```
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_parameters (in ParameterDef old_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_parameters_at (in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  StringList contexts ()
    raises (Reflective::SemanticError);
  void set_contexts (in StringList new_value)
    raises (Reflective::SemanticError);
  void unset_contexts ()
    raises (Reflective::SemanticError);
  void add_contexts (in string new_value)
    raises (Reflective::SemanticError);
  void add_contexts_before (
    in string new_value,
    in string before_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_contexts_at (
    in string new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_contexts (
    in string old_value,
    in string new_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_contexts_at (
    in string new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_contexts (in string old_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_contexts_at (in unsigned long position)
    raises (
      Reflective::BadPosition,
```

```
        Reflective::SemanticError);
    ExceptionDefSet exception_def ()
      raises (Reflective::SemanticError);
    void set_exception_def (in ExceptionDefSet new_value)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void unset_exception_def ()
      raises (Reflective::SemanticError);
    void add_exception_def (in ExceptionDef new_value)
      raises (Reflective::StructuralError);
    void modify_exception_def (
      in ExceptionDef old_value,
      in ExceptionDef new_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove_exception_def (in ExceptionDef old_value)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface OperationDef

  interface ExceptionDefClass : ContainedClass
  {
    readonly attribute ExceptionDefUList all_of_kind_exception_def;
    readonly attribute ExceptionDefUList all_of_type_exception_def;
    ExceptionDef create_exception_def (
      in string identifier,
      in string repository_id,
      in string version,
      in FieldUList members)
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface ExceptionDef : ExceptionDefClass, Contained
  {
    TypeCode type_code ()
      raises (Reflective::SemanticError);
    FieldUList members ()
      raises (Reflective::SemanticError);
    void set_members (in FieldUList new_value)
      raises (Reflective::SemanticError);
    void unset_members ()
      raises (Reflective::SemanticError);
    void add_members (in Field new_value)
      raises (Reflective::SemanticError);
```

```
  void add_members_before (
    in Field new_value,
    in Field before_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void add_members_at (
    in Field new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void modify_members (
    in Field old_value,
    in Field new_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_members_at (
    in Field new_value,
    in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
  void remove_members (in Field old_value)
    raises (
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_members_at (in unsigned long position)
    raises (
      Reflective::BadPosition,
      Reflective::SemanticError);
}; // end of interface ExceptionDef

interface AttributeDefClass : TypedClass, ContainedClass
{
  readonly attribute AttributeDefUList all_of_kind_attribute_def;
  readonly attribute AttributeDefUList all_of_type_attribute_def;
  AttributeDef create_attribute_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_readonly)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface AttributeDef : AttributeDefClass, Typed, Contained
{
  boolean is_readonly ()
```

```
    raises (Reflective::SemanticError);
  void set_is_readonly (in boolean new_value)
    raises (Reflective::SemanticError);
  ExceptionDefSet set_exception ()
    raises (Reflective::SemanticError);
  void set_set_exception (in ExceptionDefSet new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void unset_set_exception ()
    raises (Reflective::SemanticError);
  void add_set_exception (in ExceptionDef new_value)
    raises (Reflective::StructuralError);
  void modify_set_exception (
    in ExceptionDef old_value,
    in ExceptionDef new_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_set_exception (in ExceptionDef old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  ExceptionDefSet get_exception ()
    raises (Reflective::SemanticError);
  void set_get_exception (in ExceptionDefSet new_value)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void unset_get_exception ()
    raises (Reflective::SemanticError);
  void add_get_exception (in ExceptionDef new_value)
    raises (Reflective::StructuralError);
  void modify_get_exception (
    in ExceptionDef old_value,
    in ExceptionDef new_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove_get_exception (in ExceptionDef old_value)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface AttributeDef

struct InterfaceDerivedFromLink
{
```

```
    InterfaceDef base;
    InterfaceDef derived;
  };
  typedef sequence<InterfaceDerivedFromLink>
InterfaceDerivedFromLinkSet;

  interface InterfaceDerivedFrom : Reflective::RefAssociation
  {
    InterfaceDerivedFromLinkSet all_interface_derived_from_links();
    boolean exists (
      in InterfaceDef base,
      in InterfaceDef derived);
    InterfaceDefSet with_base (
      in InterfaceDef base);
    InterfaceDefSet with_derived (
      in InterfaceDef derived);
    void add (
      in InterfaceDef base,
      in InterfaceDef derived)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_base (
      in InterfaceDef base,
      in InterfaceDef derived,
      in InterfaceDef new_base)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_derived (
      in InterfaceDef base,
      in InterfaceDef derived,
      in InterfaceDef new_derived)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in InterfaceDef base,
      in InterfaceDef derived)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface InterfaceDerivedFrom

  struct DiscriminatedByLink
  {
    IDLType discriminator_type;
    UnionDef union_def;
```

```
};
typedef sequence<DiscriminatedByLink> DiscriminatedByLinkSet;

interface DiscriminatedBy : Reflective::RefAssociation
{
  DiscriminatedByLinkSet all_discriminated_by_links();
  boolean exists (
    in IDLType discriminator_type,
    in UnionDef union_def);
  UnionDefSet with_discriminator_type (
    in IDLType discriminator_type);
  IDLType with_union_def (
    in UnionDef union_def);
  void add (
    in IDLType discriminator_type,
    in UnionDef union_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_discriminator_type (
    in IDLType discriminator_type,
    in UnionDef union_def,
    in IDLType new_discriminator_type)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_union_def (
    in IDLType discriminator_type,
    in UnionDef union_def,
    in UnionDef new_union_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in IDLType discriminator_type,
    in UnionDef union_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface DiscriminatedBy

struct TypedByLink
{
  IDLType idl_type;
  Typed typed;
};
typedef sequence<TypedByLink> TypedByLinkSet;
```

```
interface TypedBy : Reflective::RefAssociation
{
  TypedByLinkSet all_typed_by_links();
  boolean exists (
    in IDLType idl_type,
    in Typed typed);
  TypedSet with_idl_type (
    in IDLType idl_type);
  IDLType with_typed (
    in Typed typed);
  void add (
    in IDLType idl_type,
    in Typed typed)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_idl_type (
    in IDLType idl_type,
    in Typed typed,
    in IDLType new_idl_type)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_typed (
    in IDLType idl_type,
    in Typed typed,
    in Typed new_typed)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in IDLType idl_type,
    in Typed typed)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface TypedBy

struct SupportsLink
{
  InterfaceDef interface_def;
  ValueDef value_def;
};
typedef sequence<SupportsLink> SupportsLinkSet;

interface Supports : Reflective::RefAssociation
{
  SupportsLinkSet all_supports_links();
```

```
    boolean exists (
      in InterfaceDef interface_def,
      in ValueDef value_def);
    ValueDefSet with_interface_def (
      in InterfaceDef interface_def);
    InterfaceDef with_value_def (
      in ValueDef value_def);
    void add (
      in InterfaceDef interface_def,
      in ValueDef value_def)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_interface_def (
      in InterfaceDef interface_def,
      in ValueDef value_def,
      in InterfaceDef new_interface_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_value_def (
      in InterfaceDef interface_def,
      in ValueDef value_def,
      in ValueDef new_value_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in InterfaceDef interface_def,
      in ValueDef value_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface Supports

  struct ValueDerivedFromLink
  {
    ValueDef base;
    ValueDef derived;
  };
  typedef sequence<ValueDerivedFromLink> ValueDerivedFromLinkSet;

  interface ValueDerivedFrom : Reflective::RefAssociation
  {
    ValueDerivedFromLinkSet all_value_derived_from_links();
    boolean exists (
      in ValueDef base,
      in ValueDef derived);
```

```
    ValueDefSet with_base (
      in ValueDef base);
    ValueDef with_derived (
      in ValueDef derived);
    void add (
      in ValueDef base,
      in ValueDef derived)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_base (
      in ValueDef base,
      in ValueDef derived,
      in ValueDef new_base)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_derived (
      in ValueDef base,
      in ValueDef derived,
      in ValueDef new_derived)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in ValueDef base,
      in ValueDef derived)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface ValueDerivedFrom

  struct AbstractDerivedFromLink
  {
    ValueDef abstract_derived;
    ValueDef abstract_base;
  };
  typedef sequence<AbstractDerivedFromLink>
AbstractDerivedFromLinkSet;

  interface AbstractDerivedFrom : Reflective::RefAssociation
  {
    AbstractDerivedFromLinkSet all_abstract_derived_from_links();
    boolean exists (
      in ValueDef abstract_derived,
      in ValueDef abstract_base);
    ValueDefSet with_abstract_derived (
      in ValueDef abstract_derived);
```

```
    ValueDefSet with_abstract_base (
      in ValueDef abstract_base);
    void add (
      in ValueDef abstract_derived,
      in ValueDef abstract_base)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_abstract_derived (
      in ValueDef abstract_derived,
      in ValueDef abstract_base,
      in ValueDef new_abstract_derived)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_abstract_base (
      in ValueDef abstract_derived,
      in ValueDef abstract_base,
      in ValueDef new_abstract_base)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in ValueDef abstract_derived,
      in ValueDef abstract_base)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface AbstractDerivedFrom

  struct SetRaisesLink
  {
    ExceptionDef set_exception;
    AttributeDef set_attribute;
  };
  typedef sequence<SetRaisesLink> SetRaisesLinkSet;

  interface SetRaises : Reflective::RefAssociation
  {
    SetRaisesLinkSet all_set_raises_links();
    boolean exists (
      in ExceptionDef set_exception,
      in AttributeDef set_attribute);
    AttributeDefSet with_set_exception (
      in ExceptionDef set_exception);
    ExceptionDefSet with_set_attribute (
      in AttributeDef set_attribute);
    void add (
```

```
    in ExceptionDef set_exception,
    in AttributeDef set_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_set_exception (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute,
    in ExceptionDef new_set_exception)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_set_attribute (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute,
    in AttributeDef new_set_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface SetRaises

struct CanRaiseLink
{
  ExceptionDef exception_def;
  OperationDef operation_def;
};
typedef sequence<CanRaiseLink> CanRaiseLinkSet;

interface CanRaise : Reflective::RefAssociation
{
  CanRaiseLinkSet all_can_raise_links();
  boolean exists (
    in ExceptionDef exception_def,
    in OperationDef operation_def);
  OperationDefSet with_exception_def (
    in ExceptionDef exception_def);
  ExceptionDefSet with_operation_def (
    in OperationDef operation_def);
  void add (
    in ExceptionDef exception_def,
    in OperationDef operation_def)
    raises (
```

```
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_exception_def (
      in ExceptionDef exception_def,
      in OperationDef operation_def,
      in ExceptionDef new_exception_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_operation_def (
      in ExceptionDef exception_def,
      in OperationDef operation_def,
      in OperationDef new_operation_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in ExceptionDef exception_def,
      in OperationDef operation_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
};// end of interface CanRaise

struct GetRaisesLink
{
  ExceptionDef get_exception;
  AttributeDef get_attribute;
};
typedef sequence<GetRaisesLink> GetRaisesLinkSet;

interface GetRaises : Reflective::RefAssociation
{
  GetRaisesLinkSet all_get_raises_links();
  boolean exists (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute);
  AttributeDefSet with_get_exception (
    in ExceptionDef get_exception);
  ExceptionDefSet with_get_attribute (
    in AttributeDef get_attribute);
  void add (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_get_exception (
```

```
    in ExceptionDef get_exception,
    in AttributeDef get_attribute,
    in ExceptionDef new_get_exception)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_get_attribute (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute,
    in AttributeDef new_get_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface GetRaises

struct ContainsLink
{
  Container defined_in;
  Contained contents;
};
typedef sequence<ContainsLink> ContainsLinkSet;

interface Contains : Reflective::RefAssociation
{
  ContainsLinkSet all_contains_links();
  boolean exists (
    in Container defined_in,
    in Contained contents);
  ContainedSet with_defined_in (
    in Container defined_in);
  Container with_contents (
    in Contained contents);
  void add (
    in Container defined_in,
    in Contained contents)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_defined_in (
    in Container defined_in,
    in Contained contents,
    in Container new_defined_in)
```

```
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_contents (
      in Container defined_in,
      in Contained contents,
      in Contained new_contents)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in Container defined_in,
      in Contained contents)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
};  // end of interface Contains

interface BaseIDLPackageFactory
{
  BaseIDLPackage create_base_idl_package ()
    raises (Reflective::SemanticError);
};

interface BaseIDLPackage : Reflective::RefPackage
{
  readonly attribute ParameterDefClass parameter_def_class_ref;
  readonly attribute ConstantDefClass constant_def_class_ref;
  readonly attribute TypedClass typed_class_ref;
  readonly attribute ModuleDefClass module_def_class_ref;
  readonly attribute TypedefDefClass typedef_def_class_ref;
  readonly attribute InterfaceDefClass interface_def_class_ref;
  readonly attribute FieldClass field_class_ref;
  readonly attribute StructDefClass struct_def_class_ref;
  readonly attribute UnionDefClass union_def_class_ref;
  readonly attribute EnumDefClass enum_def_class_ref;
  readonly attribute AliasDefClass alias_def_class_ref;
  readonly attribute IDLTypeClass idltype_class_ref;
  readonly attribute StringDefClass string_def_class_ref;
  readonly attribute WstringDefClass wstring_def_class_ref;
  readonly attribute FixedDefClass fixed_def_class_ref;
  readonly attribute SequenceDefClass sequence_def_class_ref;
  readonly attribute ArrayDefClass array_def_class_ref;
  readonly attribute PrimitiveDefClass primitive_def_class_ref;
  readonly attribute UnionFieldClass union_field_class_ref;
  readonly attribute ContainerClass container_class_ref;
  readonly attribute ValueMemberDefClass value_member_def_class_ref;
  readonly attribute ValueDefClass value_def_class_ref;
```

```
    readonly attribute ValueBoxDefClass value_box_def_class_ref;
    readonly attribute OperationDefClass operation_def_class_ref;
    readonly attribute ExceptionDefClass exception_def_class_ref;
    readonly attribute ContainedClass contained_class_ref;
    readonly attribute AttributeDefClass attribute_def_class_ref;
    readonly attribute InterfaceDerivedFrom interface_derived_from_ref;
    readonly attribute DiscriminatedBy discriminated_by_ref;
    readonly attribute TypedBy typed_by_ref;
    readonly attribute Supports supports_ref;
    readonly attribute ValueDerivedFrom value_derived_from_ref;
    readonly attribute AbstractDerivedFrom abstract_derived_from_ref;
    readonly attribute SetRaises set_raises_ref;
    readonly attribute CanRaise can_raise_ref;
    readonly attribute GetRaises get_raises_ref;
    readonly attribute Contains contains_ref;
  };
};


module ComponentIDL
{

  interface ComponentDefClass;
  interface ComponentDef;
  typedef sequence<ComponentDef> ComponentDefSet;
  typedef sequence<ComponentDef> ComponentDefUList;
  interface ProvidesDefClass;
  interface ProvidesDef;
  typedef sequence<ProvidesDef> ProvidesDefSet;
  typedef sequence<ProvidesDef> ProvidesDefUList;
  interface HomeDefClass;
  interface HomeDef;
  typedef sequence<HomeDef> HomeDefSet;
  typedef sequence<HomeDef> HomeDefUList;
  interface FactoryDefClass;
  interface FactoryDef;
  typedef sequence<FactoryDef> FactoryDefUList;
  interface FinderDefClass;
  interface FinderDef;
  typedef sequence<FinderDef> FinderDefUList;
  interface EmitsDefClass;
  interface EmitsDef;
  typedef sequence<EmitsDef> EmitsDefSet;
  typedef sequence<EmitsDef> EmitsDefUList;
  interface ConsumesDefClass;
  interface ConsumesDef;
  typedef sequence<ConsumesDef> ConsumesDefSet;
  typedef sequence<ConsumesDef> ConsumesDefUList;
  interface PrimaryKeyDefClass;
  interface PrimaryKeyDef;
  typedef sequence<PrimaryKeyDef> PrimaryKeyDefSet;
```

```
typedef sequence<PrimaryKeyDef> PrimaryKeyDefUList;
interface UsesDefClass;
interface UsesDef;
typedef sequence<UsesDef> UsesDefSet;
typedef sequence<UsesDef> UsesDefUList;
interface PublishesDefClass;
interface PublishesDef;
typedef sequence<PublishesDef> PublishesDefSet;
typedef sequence<PublishesDef> PublishesDefUList;
interface ComponentIDLPackage;

interface ComponentDefClass : BaseIDL::InterfaceDefClass
{
  readonly attribute ComponentDefUList all_of_kind_component_def;
  readonly attribute ComponentDefUList all_of_type_component_def;
  ComponentDef create_component_def ()
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ComponentDef : ComponentDefClass, BaseIDL::InterfaceDef
{
  BaseIDL::InterfaceDef supports ()
    raises (
      Reflective::NotSet,
      Reflective::SemanticError);
  void set_supports (in BaseIDL::InterfaceDef new_value)
    raises (Reflective::SemanticError);
  void unset_supports ()
    raises (Reflective::SemanticError);
}; // end of interface ComponentDef

interface ProvidesDefClass : BaseIDL::ContainedClass
{
  readonly attribute ProvidesDefUList all_of_kind_provides_def;
  readonly attribute ProvidesDefUList all_of_type_provides_def;
  ProvidesDef create_provides_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ProvidesDef : ProvidesDefClass, BaseIDL::Contained
{
  BaseIDL::InterfaceDef uml_interface ()
    raises (Reflective::SemanticError);
```

```
    void set_uml_interface (in BaseIDL::InterfaceDef new_value)
      raises (Reflective::SemanticError);
  }; // end of interface ProvidesDef

  interface HomeDefClass : BaseIDL::InterfaceDefClass
  {
    readonly attribute HomeDefUList all_of_kind_home_def;
    readonly attribute HomeDefUList all_of_type_home_def;
    HomeDef create_home_def ()
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface HomeDef : HomeDefClass, BaseIDL::InterfaceDef
  {
    ComponentDef manages ()
      raises (Reflective::SemanticError);
    void set_manages (in ComponentDef new_value)
      raises (Reflective::SemanticError);
  }; // end of interface HomeDef

  interface FactoryDefClass : BaseIDL::OperationDefClass
  {
    readonly attribute FactoryDefUList all_of_kind_factory_def;
    readonly attribute FactoryDefUList all_of_type_factory_def;
    FactoryDef create_factory_def (
      in string identifier,
      in string repository_id,
      in string version,
      in string absolute_name)
      raises (
        Reflective::SemanticError,
        Reflective::ConstraintError);
  };

  interface FactoryDef : FactoryDefClass, BaseIDL::OperationDef
  {
  }; // end of interface FactoryDef

  interface FinderDefClass : BaseIDL::OperationDefClass
  {
    readonly attribute FinderDefUList all_of_kind_finder_def;
    readonly attribute FinderDefUList all_of_type_finder_def;
    FinderDef create_finder_def (
      in string identifier,
      in string repository_id,
      in string version,
      in string absolute_name)
      raises (
        Reflective::SemanticError,
```

```
        Reflective::ConstraintError);
};

interface FinderDef : FinderDefClass, BaseIDL::OperationDef
{
}; // end of interface FinderDef

interface EmitsDefClass : BaseIDL::ContainedClass
{
  readonly attribute EmitsDefUList all_of_kind_emits_def;
  readonly attribute EmitsDefUList all_of_type_emits_def;
  EmitsDef create_emits_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface EmitsDef : EmitsDefClass, BaseIDL::Contained
{
  BaseIDL::ValueDef event ()
    raises (Reflective::SemanticError);
  void set_event (in BaseIDL::ValueDef new_value)
    raises (Reflective::SemanticError);
}; // end of interface EmitsDef

interface ConsumesDefClass : BaseIDL::ContainedClass
{
  readonly attribute ConsumesDefUList all_of_kind_consumes_def;
  readonly attribute ConsumesDefUList all_of_type_consumes_def;
  ConsumesDef create_consumes_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface ConsumesDef : ConsumesDefClass, BaseIDL::Contained
{
  BaseIDL::ValueDef event ()
    raises (Reflective::SemanticError);
  void set_event (in BaseIDL::ValueDef new_value)
    raises (Reflective::SemanticError);
}; // end of interface ConsumesDef
```

```
interface PrimaryKeyDefClass : BaseIDL::ContainedClass
{
  readonly attribute PrimaryKeyDefUList all_of_kind_primary_key_def;
  readonly attribute PrimaryKeyDefUList all_of_type_primary_key_def;
  PrimaryKeyDef create_primary_key_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface PrimaryKeyDef : PrimaryKeyDefClass, BaseIDL::Contained
{
  BaseIDL::ValueDef type ()
    raises (Reflective::SemanticError);
  void set_type (in BaseIDL::ValueDef new_value)
    raises (Reflective::SemanticError);
}; // end of interface PrimaryKeyDef

interface UsesDefClass : BaseIDL::ContainedClass
{
  readonly attribute UsesDefUList all_of_kind_uses_def;
  readonly attribute UsesDefUList all_of_type_uses_def;
  UsesDef create_uses_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name,
    in boolean multiple)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface UsesDef : UsesDefClass, BaseIDL::Contained
{
  boolean multiple ()
    raises (Reflective::SemanticError);
  void set_multiple (in boolean new_value)
    raises (Reflective::SemanticError);
  BaseIDL::InterfaceDef uml_interface ()
    raises (Reflective::SemanticError);
  void set_uml_interface (in BaseIDL::InterfaceDef new_value)
    raises (Reflective::SemanticError);
}; // end of interface UsesDef

interface PublishesDefClass : BaseIDL::ContainedClass
{
```

```
  readonly attribute PublishesDefUList all_of_kind_publishes_def;
  readonly attribute PublishesDefUList all_of_type_publishes_def;
  PublishesDef create_publishes_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string absolute_name)
    raises (
      Reflective::SemanticError,
      Reflective::ConstraintError);
};

interface PublishesDef : PublishesDefClass, BaseIDL::Contained
{
  BaseIDL::ValueDef event ()
    raises (Reflective::SemanticError);
  void set_event (in BaseIDL::ValueDef new_value)
    raises (Reflective::SemanticError);
}; // end of interface PublishesDef

struct ProvidesInterfaceLink
{
  BaseIDL::InterfaceDef uml_interface;
  ProvidesDef provides_def;
};
typedef sequence<ProvidesInterfaceLink> ProvidesInterfaceLinkSet;

interface ProvidesInterface : Reflective::RefAssociation
{
  ProvidesInterfaceLinkSet all_provides_interface_links();
  boolean exists (
    in BaseIDL::InterfaceDef uml_interface,
    in ProvidesDef provides_def);
  ProvidesDefSet with_uml_interface (
    in BaseIDL::InterfaceDef uml_interface);
  BaseIDL::InterfaceDef with_provides_def (
    in ProvidesDef provides_def);
  void add (
    in BaseIDL::InterfaceDef uml_interface,
    in ProvidesDef provides_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_uml_interface (
    in BaseIDL::InterfaceDef uml_interface,
    in ProvidesDef provides_def,
    in BaseIDL::InterfaceDef new_uml_interface)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
```

```
    void modify_provides_def (
      in BaseIDL::InterfaceDef uml_interface,
      in ProvidesDef provides_def,
      in ProvidesDef new_provides_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in BaseIDL::InterfaceDef uml_interface,
      in ProvidesDef provides_def)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface ProvidesInterface

  struct EmitsEventLink
  {
    BaseIDL::ValueDef event;
    EmitsDef emits_def;
  };
  typedef sequence<EmitsEventLink> EmitsEventLinkSet;

  interface EmitsEvent : Reflective::RefAssociation
  {
    EmitsEventLinkSet all_emits_event_links();
    boolean exists (
      in BaseIDL::ValueDef event,
      in EmitsDef emits_def);
    EmitsDefSet with_event (
      in BaseIDL::ValueDef event);
    BaseIDL::ValueDef with_emits_def (
      in EmitsDef emits_def);
    void add (
      in BaseIDL::ValueDef event,
      in EmitsDef emits_def)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_event (
      in BaseIDL::ValueDef event,
      in EmitsDef emits_def,
      in BaseIDL::ValueDef new_event)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_emits_def (
      in BaseIDL::ValueDef event,
      in EmitsDef emits_def,
```

```
    in EmitsDef new_emits_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in BaseIDL::ValueDef event,
    in EmitsDef emits_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface EmitsEvent

struct ConsumesEventLink
{
  BaseIDL::ValueDef event;
  ConsumesDef consumes_def;
};
typedef sequence<ConsumesEventLink> ConsumesEventLinkSet;

interface ConsumesEvent : Reflective::RefAssociation
{
  ConsumesEventLinkSet all_consumes_event_links();
  boolean exists (
    in BaseIDL::ValueDef event,
    in ConsumesDef consumes_def);
  ConsumesDefSet with_event (
    in BaseIDL::ValueDef event);
  BaseIDL::ValueDef with_consumes_def (
    in ConsumesDef consumes_def);
  void add (
    in BaseIDL::ValueDef event,
    in ConsumesDef consumes_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_event (
    in BaseIDL::ValueDef event,
    in ConsumesDef consumes_def,
    in BaseIDL::ValueDef new_event)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_consumes_def (
    in BaseIDL::ValueDef event,
    in ConsumesDef consumes_def,
    in ConsumesDef new_consumes_def)
    raises (
      Reflective::StructuralError,
```

```
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in BaseIDL::ValueDef event,
    in ConsumesDef consumes_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface ConsumesEvent

struct UsesInterfaceLink
{
  BaseIDL::InterfaceDef uml_interface;
  UsesDef uses_def;
};
typedef sequence<UsesInterfaceLink> UsesInterfaceLinkSet;

interface UsesInterface : Reflective::RefAssociation
{
  UsesInterfaceLinkSet all_uses_interface_links();
  boolean exists (
    in BaseIDL::InterfaceDef uml_interface,
    in UsesDef uses_def);
  UsesDefSet with_uml_interface (
    in BaseIDL::InterfaceDef uml_interface);
  BaseIDL::InterfaceDef with_uses_def (
    in UsesDef uses_def);
  void add (
    in BaseIDL::InterfaceDef uml_interface,
    in UsesDef uses_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_uml_interface (
    in BaseIDL::InterfaceDef uml_interface,
    in UsesDef uses_def,
    in BaseIDL::InterfaceDef new_uml_interface)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_uses_def (
    in BaseIDL::InterfaceDef uml_interface,
    in UsesDef uses_def,
    in UsesDef new_uses_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
```

```
    in BaseIDL::InterfaceDef uml_interface,
    in UsesDef uses_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface UsesInterface

struct KeyTypeLink
{
  BaseIDL::ValueDef type;
  PrimaryKeyDef key_def;
};
typedef sequence<KeyTypeLink> KeyTypeLinkSet;

interface KeyType : Reflective::RefAssociation
{
  KeyTypeLinkSet all_key_type_links();
  boolean exists (
    in BaseIDL::ValueDef type,
    in PrimaryKeyDef key_def);
  PrimaryKeyDefSet with_type (
    in BaseIDL::ValueDef type);
  BaseIDL::ValueDef with_key_def (
    in PrimaryKeyDef key_def);
  void add (
    in BaseIDL::ValueDef type,
    in PrimaryKeyDef key_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_type (
    in BaseIDL::ValueDef type,
    in PrimaryKeyDef key_def,
    in BaseIDL::ValueDef new_type)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_key_def (
    in BaseIDL::ValueDef type,
    in PrimaryKeyDef key_def,
    in PrimaryKeyDef new_key_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in BaseIDL::ValueDef type,
    in PrimaryKeyDef key_def)
    raises (
```

```
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
};// end of interface KeyType

struct ComponentHomeLink
{
  ComponentDef manages;
  HomeDef home;
};
typedef sequence<ComponentHomeLink> ComponentHomeLinkSet;

interface ComponentHome : Reflective::RefAssociation
{
  ComponentHomeLinkSet all_component_home_links();
  boolean exists (
    in ComponentDef manages,
    in HomeDef home);
  HomeDefSet with_manages (
    in ComponentDef manages);
  ComponentDef with_home (
    in HomeDef home);
  void add (
    in ComponentDef manages,
    in HomeDef home)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_manages (
    in ComponentDef manages,
    in HomeDef home,
    in ComponentDef new_manages)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_home (
    in ComponentDef manages,
    in HomeDef home,
    in HomeDef new_home)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in ComponentDef manages,
    in HomeDef home)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
```

```
  }; // end of interface ComponentHome

  struct ComponentSupportsLink
  {
    BaseIDL::InterfaceDef supports;
    ComponentDef components;
  };
typedef sequence<ComponentSupportsLink> ComponentSupportsLinkSet;

  interface ComponentSupports : Reflective::RefAssociation
  {
    ComponentSupportsLinkSet all_component_supports_links();
    boolean exists (
      in BaseIDL::InterfaceDef supports,
      in ComponentDef components);
    ComponentDefSet with_supports (
      in BaseIDL::InterfaceDef supports);
    BaseIDL::InterfaceDef with_components (
      in ComponentDef components);
    void add (
      in BaseIDL::InterfaceDef supports,
      in ComponentDef components)
      raises (
        Reflective::StructuralError,
        Reflective::SemanticError);
    void modify_supports (
      in BaseIDL::InterfaceDef supports,
      in ComponentDef components,
      in BaseIDL::InterfaceDef new_supports)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void modify_components (
      in BaseIDL::InterfaceDef supports,
      in ComponentDef components,
      in ComponentDef new_components)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
    void remove (
      in BaseIDL::InterfaceDef supports,
      in ComponentDef components)
      raises (
        Reflective::StructuralError,
        Reflective::NotFound,
        Reflective::SemanticError);
  }; // end of interface ComponentSupports

  struct PublishesEventLink
```

```
{
  BaseIDL::ValueDef event;
  PublishesDef publishes_def;
};
typedef sequence<PublishesEventLink> PublishesEventLinkSet;

interface PublishesEvent : Reflective::RefAssociation
{
  PublishesEventLinkSet all_publishes_event_links();
  boolean exists (
    in BaseIDL::ValueDef event,
    in PublishesDef publishes_def);
  PublishesDefSet with_event (
    in BaseIDL::ValueDef event);
  BaseIDL::ValueDef with_publishes_def (
    in PublishesDef publishes_def);
  void add (
    in BaseIDL::ValueDef event,
    in PublishesDef publishes_def)
    raises (
      Reflective::StructuralError,
      Reflective::SemanticError);
  void modify_event (
    in BaseIDL::ValueDef event,
    in PublishesDef publishes_def,
    in BaseIDL::ValueDef new_event)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void modify_publishes_def (
    in BaseIDL::ValueDef event,
    in PublishesDef publishes_def,
    in PublishesDef new_publishes_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
  void remove (
    in BaseIDL::ValueDef event,
    in PublishesDef publishes_def)
    raises (
      Reflective::StructuralError,
      Reflective::NotFound,
      Reflective::SemanticError);
}; // end of interface PublishesEvent

interface ComponentIDLPackageFactory
{
  ComponentIDLPackage create_component_idl_package ()
    raises (Reflective::SemanticError);
```

```
  };

  interface ComponentIDLPackage : Reflective::RefPackage
  {
    readonly attribute ComponentDefClass component_def_class_ref;
    readonly attribute ProvidesDefClass provides_def_class_ref;
    readonly attribute HomeDefClass home_def_class_ref;
    readonly attribute FactoryDefClass factory_def_class_ref;
    readonly attribute FinderDefClass finder_def_class_ref;
    readonly attribute EmitsDefClass emits_def_class_ref;
    readonly attribute ConsumesDefClass consumes_def_class_ref;
    readonly attribute PrimaryKeyDefClass primary_key_def_class_ref;
    readonly attribute UsesDefClass uses_def_class_ref;
    readonly attribute PublishesDefClass publishes_def_class_ref;
    readonly attribute ProvidesInterface provides_interface_ref;
    readonly attribute EmitsEvent emits_event_ref;
    readonly attribute ConsumesEvent consumes_event_ref;
    readonly attribute UsesInterface uses_interface_ref;
    readonly attribute KeyType key_type_ref;
    readonly attribute ComponentHome component_home_ref;
    readonly attribute ComponentSupports component_supports_ref;
    readonly attribute PublishesEvent publishes_event_ref;
  };
};
```

C.2 Packaging and Deployment Metamodel

C.2.1 XMI DTD

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI is the top-level XML element for XMI transfer text       -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI (XMI.header, XMI.content?, XMI.difference*,
               XMI.extensions*) >
<!ATTLIST XMI
            xmi.version CDATA #FIXED "1.0"
            timestamp CDATA #IMPLIED
            verified (true | false) #IMPLIED
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.header contains documentation and identifies the model,  -->
<!-- metamodel, and metametamodel                                 -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.header (XMI.documentation?, XMI.model*, XMI.metamodel*,
                      XMI.metametamodel*) >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- documentation for transfer data                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.documentation (#PCDATA | XMI.owner | XMI.contact |
                             XMI.longDescription | XMI.shortDescription |
                             XMI.exporter | XMI.exporterVersion |
                             XMI.notice)* >

<!ELEMENT XMI.owner ANY >

<!ELEMENT XMI.contact ANY >

<!ELEMENT XMI.longDescription ANY >

<!ELEMENT XMI.shortDescription ANY >

<!ELEMENT XMI.exporter ANY >

<!ELEMENT XMI.exporterVersion ANY >

<!ELEMENT XMI.exporterID ANY >

<!ELEMENT XMI.notice ANY >
```

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.element.att defines the attributes that each XML element -->
<!-- that corresponds to a metamodel class must have to conform to -->
<!-- the XMI specification.                                       -->
<!-- ____________________________________________________________ -->

<!ENTITY % XMI.element.att
               'xmi.id ID #IMPLIED xmi.label CDATA #IMPLIED xmi.uuid
                CDATA #IMPLIED ' >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.link.att defines the attributes that each XML element that -->
<!-- corresponds to a metamodel class must have to enable it to   -->
<!-- function as a simple XLink as well as refer to model         -->
<!-- constructs within the same XMI file.                         -->
<!-- ____________________________________________________________ -->

<!ENTITY % XMI.link.att
               'xml:link CDATA #IMPLIED inline (true | false) #IMPLIED
                actuate (show | user) #IMPLIED href CDATA #IMPLIED role
                CDATA #IMPLIED title CDATA #IMPLIED show (embed | replace
                | new) #IMPLIED behavior CDATA #IMPLIED xmi.idref IDREF
                #IMPLIED xmi.uuidref CDATA #IMPLIED' >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.model identifies the model(s) being transferred          -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.model ANY >
<!ATTLIST XMI.model
           %XMI.link.att;
           xmi.name     CDATA #REQUIRED
           xmi.version  CDATA #IMPLIED
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.metamodel identifies the metamodel(s) for the transferred -->
<!-- data                                                         -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.metamodel ANY >
<!ATTLIST XMI.metamodel
           %XMI.link.att;
           xmi.name     CDATA #REQUIRED
           xmi.version  CDATA #IMPLIED
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.metametamodel identifies the metametamodel(s) for the    -->
```

```
<!-- transferred data                                                    -->
<!-- ________________________________________________________________ -->

<!ELEMENT XMI.metametamodel ANY >
<!ATTLIST XMI.metametamodel
            %XMI.link.att;
            xmi.name     CDATA #REQUIRED
            xmi.version  CDATA #IMPLIED
>

<!-- ________________________________________________________________ -->
<!--                                                                  -->
<!-- XMI.content is the actual data being transferred                 -->
<!-- ________________________________________________________________ -->

<!ELEMENT XMI.content ANY >

<!-- ________________________________________________________________ -->
<!--                                                                  -->
<!-- XMI.extensions contains data to transfer that does not conform  -->
<!-- to the metamodel(s) in the header                                -->
<!-- ________________________________________________________________ -->

<!ELEMENT XMI.extensions ANY >
<!ATTLIST XMI.extensions
            xmi.extender CDATA #REQUIRED
>

<!-- ________________________________________________________________ -->
<!--                                                                  -->
<!-- extension contains information related to a specific model       -->
<!-- construct that is not defined in the metamodel(s) in the header  -->
<!-- ________________________________________________________________ -->

<!ELEMENT XMI.extension ANY >
<!ATTLIST XMI.extension
            %XMI.element.att;
            %XMI.link.att;
            xmi.extender   CDATA #REQUIRED
            xmi.extenderID CDATA #REQUIRED
>

<!-- ________________________________________________________________ -->
<!--                                                                  -->
<!-- XMI.difference holds XML elements representing differences to a  -->
<!-- base model                                                       -->
<!-- ________________________________________________________________ -->

<!ELEMENT XMI.difference (XMI.difference | XMI.delete | XMI.add |
                          XMI.replace)* >
<!ATTLIST XMI.difference
            %XMI.element.att;
            %XMI.link.att;
>
```

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.delete represents a deletion from a base model           -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.delete EMPTY >
<!ATTLIST XMI.delete
            %XMI.element.att;
            %XMI.link.att;
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.add represents an addition to a base model               -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.add ANY >
<!ATTLIST XMI.add
            %XMI.element.att;
            %XMI.link.att;
            xmi.position CDATA "-1"
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.replace represents the replacement of a model construct  -->
<!-- with another model construct in a base model                 -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.replace ANY >
<!ATTLIST XMI.replace
            %XMI.element.att;
            %XMI.link.att;
            xmi.position CDATA "-1"
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.reference may be used to refer to data types not defined in -->
<!-- the metamodel                                                -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
            %XMI.link.att;
>

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- This section contains the declaration of XML elements        -->
<!-- representing data types                                      -->
<!-- ____________________________________________________________ -->

<!ELEMENT XMI.TypeDefinitions ANY >
```

```
<!ELEMENT XMI.field ANY >

<!ELEMENT XMI.seqItem ANY >

<!ELEMENT XMI.octetStream (#PCDATA) >

<!ELEMENT XMI.unionDiscrim ANY >

<!ELEMENT XMI.enum EMPTY >
<!ATTLIST XMI.enum
            xmi.value CDATA #REQUIRED
>

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
            %XMI.link.att;
            xmi.type CDATA #IMPLIED
            xmi.name CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTypeCode (XMI.CorbaTcAlias | XMI.CorbaTcStruct |
                             XMI.CorbaTcSequence | XMI.CorbaTcArray |
                             XMI.CorbaTcEnum | XMI.CorbaTcUnion |
                             XMI.CorbaTcExcept | XMI.CorbaTcString |
                             XMI.CorbaTcWstring | XMI.CorbaTcShort |
                             XMI.CorbaTcLong | XMI.CorbaTcUshort |
                             XMI.CorbaTcUlong | XMI.CorbaTcFloat |
                             XMI.CorbaTcDouble | XMI.CorbaTcBoolean |
                             XMI.CorbaTcChar | XMI.CorbaTcWchar |
                             XMI.CorbaTcOctet | XMI.CorbaTcAny |
                             XMI.CorbaTcTypeCode | XMI.CorbaTcPrincipal |
                             XMI.CorbaTcNull | XMI.CorbaTcVoid |
                             XMI.CorbaTcLongLong |
                             XMI.CorbaTcLongDouble) >
<!ATTLIST XMI.CorbaTypeCode
            %XMI.element.att;
>

<!ELEMENT XMI.CorbaTcAlias (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcAlias
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcStruct (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcStruct
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcField (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcField
            xmi.tcName CDATA #REQUIRED
>
```

```
<!ELEMENT XMI.CorbaTcSequence (XMI.CorbaTypeCode |
                               XMI.CorbaRecursiveType) >
<!ATTLIST XMI.CorbaTcSequence
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaRecursiveType EMPTY >
<!ATTLIST XMI.CorbaRecursiveType
            xmi.offset CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcArray (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcArray
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcObjRef EMPTY >
<!ATTLIST XMI.CorbaTcObjRef
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcEnum (XMI.CorbaTcEnumLabel) >
<!ATTLIST XMI.CorbaTcEnum
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcEnumLabel EMPTY >
<!ATTLIST XMI.CorbaTcEnumLabel
            xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnionMbr (XMI.CorbaTypeCode, XMI.any) >
<!ATTLIST XMI.CorbaTcUnionMbr
            xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnion (XMI.CorbaTypeCode, XMI.CorbaTcUnionMbr*) >
<!ATTLIST XMI.CorbaTcUnion
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcExcept (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcExcept
            xmi.tcName CDATA #REQUIRED
            xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcString EMPTY >
<!ATTLIST XMI.CorbaTcString
            xmi.tcLength CDATA #REQUIRED
>
```

```
<!ELEMENT XMI.CorbaTcWstring EMPTY >
<!ATTLIST XMI.CorbaTcWstring
            xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcFixed EMPTY >
<!ATTLIST XMI.CorbaTcFixed
            xmi.tcDigits CDATA #REQUIRED
            xmi.tcScale  CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcShort EMPTY >

<!ELEMENT XMI.CorbaTcLong EMPTY >

<!ELEMENT XMI.CorbaTcUshort EMPTY >

<!ELEMENT XMI.CorbaTcUlong EMPTY >

<!ELEMENT XMI.CorbaTcFloat EMPTY >

<!ELEMENT XMI.CorbaTcDouble EMPTY >

<!ELEMENT XMI.CorbaTcBoolean EMPTY >

<!ELEMENT XMI.CorbaTcChar EMPTY >

<!ELEMENT XMI.CorbaTcWchar EMPTY >

<!ELEMENT XMI.CorbaTcOctet EMPTY >

<!ELEMENT XMI.CorbaTcAny EMPTY >

<!ELEMENT XMI.CorbaTcTypeCode EMPTY >

<!ELEMENT XMI.CorbaTcPrincipal EMPTY >

<!ELEMENT XMI.CorbaTcNull EMPTY >

<!ELEMENT XMI.CorbaTcVoid EMPTY >

<!ELEMENT XMI.CorbaTcLongLong EMPTY >

<!ELEMENT XMI.CorbaTcLongDouble EMPTY >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE:                                           -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Softpkg.ThreadSafetyKind
           ' XMI.value ( none|class|instance) #REQUIRED'>
```

```
<!ENTITY % Softpkg.ActionKind
            ' XMI.value ( assert|install) #REQUIRED'>

<!ENTITY % Component.ComponentKind
            ' XMI.value ( service|session|process|entity|unclassified)
#REQUIRED'>

<!ENTITY % Component.TransactionSupportKind
            ' XMI.value (
notSupported|required|supports|requiresNew|mandatory|never) #REQUIRED'>

<!ENTITY % Component.SecurityCredentialKind
            ' XMI.value ( client|system|specified) #REQUIRED'>

<!ENTITY % Component.ContainerThreadingKind
            ' XMI.value ( serialize|multithread) #REQUIRED'>

<!ENTITY % Component.EventPolicyKind
            ' XMI.value ( normal|default|transaction) #REQUIRED'>

<!ENTITY % Component.LifeTimeKind
            ' XMI.value ( process|method|transaction) #REQUIRED'>

<!ENTITY % Component.InterfacePortKind
            ' XMI.value ( provides|uses) #REQUIRED'>

<!ENTITY % Component.EventPortKind
            ' XMI.value ( emits|publishes|consumes) #REQUIRED'>

<!ENTITY % Component.PersistenceResponsibilityKind
            ' XMI.value ( container|component) #REQUIRED'>

<!ENTITY % Component.POAIdAssignmentPolicy
            ' XMI.value ( USER_ID|SYSTEM_ID) #REQUIRED'>

<!ENTITY % Component.POAIdUniquenessPolicy
            ' XMI.value ( UNIQUE_ID|MULTIPLE_ID) #REQUIRED'>

<!ENTITY % Component.POAImplicitActivationPolicy
            ' XMI.value ( IMPLICIT_ACTIVATION|NON_IMPLICIT_ACTIVATION)
#REQUIRED'>

<!ENTITY % Component.POALifeSpanPolicy
            ' XMI.value ( TRANSIENT|PERSISTENT) #REQUIRED'>

<!ENTITY % Component.POARequestProcessingPolicy
            ' XMI.value (
USE_ACTIVE_OBJECT_MAP_ONLY|USE_DEFAULT_SERVANT|USE_SERVANT_MANAGER)
#REQUIRED'>

<!ENTITY % Component.POAServantRetentionPolicy
            ' XMI.value ( RETAIN|NON_RETAIN) #REQUIRED'>

<!ENTITY % Component.POAThreadPolicy
            ' XMI.value ( ORB_CTRL_MODEL|SINGLE_THREAD_SAFE) #REQUIRED'>
```

```
<!ENTITY % PropertySet.SimpleType
          ' XMI.value (
boolean|char|double|float|short|long|objectReference|octet|short|string|u
long|ushort) #REQUIRED'>


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE: PDGeneral                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->


<!-- *****    PDGeneral.FileInArchive_Link   *******  -->

<!ELEMENT PDGeneral.FileInArchive.link ( PDGeneral.Link)? >


<!-- *****    PDGeneral.Struct_Simple   *******  -->

<!ELEMENT PDGeneral.Struct_Simple.struct (XMI.reference )  >

<!ELEMENT PDGeneral.Struct_Simple.simple (XMI.reference )* >


<!-- *****    PDGeneral.Struct_Sequence   *******  -->

<!ELEMENT PDGeneral.Struct_Sequence.struct (XMI.reference )  >

<!ELEMENT PDGeneral.Struct_Sequence.sequence (XMI.reference )* >


<!-- *****    PDGeneral.Struct_Struct   *******  -->

<!ELEMENT PDGeneral.Struct_Struct.containedIn (XMI.reference )  >

<!ELEMENT PDGeneral.Struct_Struct.struct (XMI.reference )* >


<!-- *****    PDGeneral.Sequence_Struct   *******  -->

<!ELEMENT PDGeneral.Sequence_Struct.sequence (XMI.reference )  >

<!ELEMENT PDGeneral.Sequence_Struct.struct (XMI.reference )* >


<!-- *****    PDGeneral.Sequence_Simple   *******  -->

<!ELEMENT PDGeneral.Sequence_Simple.sequence (XMI.reference )  >

<!ELEMENT PDGeneral.Sequence_Simple.simple (XMI.reference )* >


<!-- *****    PDGeneral.Seqence_Sequence   *******  -->
```

```
<!ELEMENT PDGeneral.Seqence_Sequence.containedIn (XMI.reference )  >

<!ELEMENT PDGeneral.Seqence_Sequence.simple (XMI.reference )* >


<!-- *****    PDGeneral.Properties_Sequence   *******  -->

<!ELEMENT PDGeneral.Properties_Sequence.properties (XMI.reference )  >

<!ELEMENT PDGeneral.Properties_Sequence.sequence (XMI.reference )* >


<!-- *****    PDGeneral.Properties_Struct   *******  -->

<!ELEMENT PDGeneral.Properties_Struct.properties (XMI.reference )  >

<!ELEMENT PDGeneral.Properties_Struct.struct (XMI.reference )* >


<!-- *****    PDGeneral.Properties_Simple   *******  -->

<!ELEMENT PDGeneral.Properties_Simple.properties (XMI.reference )  >

<!ELEMENT PDGeneral.Properties_Simple.simple (XMI.reference )* >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: PDGeneral.CodeBase                    -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT PDGeneral.CodeBase.filename (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.CodeBase.href (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.CodeBaseProperties '(PDGeneral.CodeBase.filename ?
   ,PDGeneral.CodeBase.href  )' >

<!ELEMENT PDGeneral.CodeBase ( %PDGeneral.CodeBaseProperties; )?>

<!ATTLIST PDGeneral.CodeBase %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: PDGeneral.Extension                   -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT PDGeneral.Extension.class (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.Extension.origin (#PCDATA|XMI.reference)*>
```

```
<!ELEMENT PDGeneral.Extension.id (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.Extension.extra (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.Extension.htmlForm (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.ExtensionProperties '(PDGeneral.Extension.class
   ,PDGeneral.Extension.origin
   ,PDGeneral.Extension.id ?
   ,PDGeneral.Extension.extra ?
   ,PDGeneral.Extension.htmlForm ?)' >

<!ELEMENT PDGeneral.Extension ( %PDGeneral.ExtensionProperties; )?>

<!ATTLIST PDGeneral.Extension %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PDGeneral.FileInArchive              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT PDGeneral.FileInArchive.name (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.FileInArchiveProperties
'(PDGeneral.FileInArchive.name  )' >

<!ENTITY % PDGeneral.FileInArchiveCompositions
'(PDGeneral.FileInArchive.link?)' >

<!ELEMENT PDGeneral.FileInArchive ( %PDGeneral.FileInArchiveProperties;
       ,  %PDGeneral.FileInArchiveCompositions; )?>

<!ATTLIST PDGeneral.FileInArchive %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PDGeneral.Link                       -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT PDGeneral.Link.href (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.LinkProperties '(PDGeneral.Link.href  )' >

<!ELEMENT PDGeneral.Link ( %PDGeneral.LinkProperties; )?>

<!ATTLIST PDGeneral.Link %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PDGeneral.LocalFile                  -->
```

```
<!--                                                                    -->
<!-- ______________________________________________________________ -->

<!ELEMENT PDGeneral.LocalFile.name (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.LocalFileProperties '(PDGeneral.LocalFile.name  )' >

<!ELEMENT PDGeneral.LocalFile ( %PDGeneral.LocalFileProperties; )?>

<!ATTLIST PDGeneral.LocalFile %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                    -->
<!-- METAMODEL CLASS: PDGeneral.Repository                 -->
<!--                                                                    -->
<!-- ______________________________________________________________ -->

<!ELEMENT PDGeneral.Repository.type (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.Repository.href (#PCDATA|XMI.reference)*>

<!ELEMENT PDGeneral.Repository.objectReference (#PCDATA|XMI.reference)*>

<!ENTITY % PDGeneral.RepositoryProperties '(PDGeneral.Repository.type ?
   ,PDGeneral.Repository.href ?
   ,PDGeneral.Repository.objectReference ?)' >

<!ELEMENT PDGeneral.Repository ( %PDGeneral.RepositoryProperties; )?>

<!ATTLIST PDGeneral.Repository %XMI.element.att; %XMI.link.att; >

<!ELEMENT PDGeneral ((PDGeneral.CodeBase
  |PDGeneral.Extension
  |PDGeneral.FileInArchive
  |PDGeneral.Link
  |PDGeneral.LocalFile
  |PDGeneral.Repository)*)>
<!ATTLIST PDGeneral %XMI.element.att; %XMI.link.att;>



<!-- ______________________________________________________________ -->
<!--                                                                    -->
<!-- METAMODEL PACKAGE: Softpkg                              -->
<!--                                                                    -->
<!-- ______________________________________________________________ -->


<!-- *****    Softpkg.Softpkg_Implementation   *******  -->

<!ELEMENT Softpkg.Softpkg.implementation ( Softpkg.Implementation)* >


<!-- *****    Softpkg.Implementation_OS   *******  -->
```

```
<!ELEMENT Softpkg.Implementation.OS ( Softpkg.OS)  >

<!-- *****    Softpkg.Implementation_ProgrammingLanguage   *******  -->

<!ELEMENT Softpkg.Implementation.programmingLanguage (
Softpkg.ProgrammingLanguage)  >

<!-- *****    Softpkg.Softpkg_IDL   *******  -->

<!ELEMENT Softpkg.Softpkg.IDL ( Softpkg.IDL)* >

<!-- *****    Softpkg.Softpkg_PropertyFile   *******  -->

<!ELEMENT Softpkg.Softpkg.propertyFile ( Softpkg.PropertyFile)* >

<!-- *****    Softpkg.Softpkg_Dependency   *******  -->

<!ELEMENT Softpkg.Softpkg.dependency ( Softpkg.Dependency)* >

<!-- *****    Softpkg.Softpkg_Descriptor   *******  -->

<!ELEMENT Softpkg.Softpkg.descriptor ( Softpkg.Descriptor)* >

<!-- *****    Softpkg.Softpkg_Extension   *******  -->

<!ELEMENT Softpkg.Softpkg.extension ( PDGeneral.Extension)* >

<!-- *****    Softpkg.Softpkg_Author   *******  -->

<!ELEMENT Softpkg.Softpkg.author ( Softpkg.Author)* >

<!-- *****    Softpkg.IDL_Link   *******  -->

<!ELEMENT Softpkg.IDL.link ( PDGeneral.Link)? >

<!-- *****    Softpkg.IDL_FileInArchive   *******  -->

<!ELEMENT Softpkg.IDL.fileInArchive ( PDGeneral.FileInArchive)? >

<!-- *****    Softpkg.IDL_Repository   *******  -->

<!ELEMENT Softpkg.IDL.repository ( PDGeneral.Repository)? >
```

```
<!-- *****    Softpkg.PropertyFile_Link   *******  -->

<!ELEMENT Softpkg.PropertyFile.link ( PDGeneral.Link)? >


<!-- *****    Softpkg.PropertyFile_FileInArchive   *******  -->

<!ELEMENT Softpkg.PropertyFile.fileInArchive ( PDGeneral.FileInArchive)?
>


<!-- *****    Softpkg.Descriptor_Link   *******  -->

<!ELEMENT Softpkg.Descriptor.link ( PDGeneral.Link)? >


<!-- *****    Softpkg.Descriptor_FileInArchive   *******  -->

<!ELEMENT Softpkg.Descriptor.fileInArchive ( PDGeneral.FileInArchive)? >


<!-- *****    Softpkg.Dependency_CodeBase   *******  -->

<!ELEMENT Softpkg.Dependency.codeBase ( PDGeneral.CodeBase)* >


<!-- *****    Softpkg.Dependency_FileInArchive   *******  -->

<!ELEMENT Softpkg.Dependency.fileInArchive ( PDGeneral.FileInArchive)* >


<!-- *****    Softpkg.Dependency_LocalFile   *******  -->

<!ELEMENT Softpkg.Dependency.localFile ( PDGeneral.LocalFile)* >


<!-- *****    Softpkg.Implementation_Code   *******  -->

<!ELEMENT Softpkg.Implementation.code ( Softpkg.Code)? >


<!-- *****    Softpkg.Code_CodeBase   *******  -->

<!ELEMENT Softpkg.Code.codeBase ( PDGeneral.CodeBase)? >


<!-- *****    Softpkg.Code_FileInArchive   *******  -->

<!ELEMENT Softpkg.Code.fileInArchive ( PDGeneral.FileInArchive)? >


<!-- *****    Softpkg.Code_Link   *******  -->

<!ELEMENT Softpkg.Code.link ( PDGeneral.Link)? >
```

```
<!-- *****    Softpkg.Code_EntryPoint   *******  -->

<!ELEMENT Softpkg.Code.entryPoint ( Softpkg.EntryPoint)? >


<!-- *****    Softpkg.Implementation_Compiler   *******  -->

<!ELEMENT Softpkg.Implementation.compiler ( Softpkg.Compiler)? >


<!-- *****    Softpkg.Implementation_Dependency   *******  -->

<!ELEMENT Softpkg.Implementation.dependency ( Softpkg.Dependency)* >


<!-- *****    Softpkg.Implementation_Extension   *******  -->

<!ELEMENT Softpkg.Implementation.extension ( PDGeneral.Extension)* >


<!-- *****    Softpkg.Implementation_Descriptor   *******  -->

<!ELEMENT Softpkg.Implementation.descriptor ( Softpkg.Descriptor)* >


<!-- *****    Softpkg.Implementation_PropertyFile   *******  -->

<!ELEMENT Softpkg.Implementation.propertyFile ( Softpkg.PropertyFile)* >


<!-- *****    Softpkg.Implementation_RunTime   *******  -->

<!ELEMENT Softpkg.Implementation.runTime ( Softpkg.RunTime)* >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.Softpkg                       -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.Softpkg.name (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Softpkg.version (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Softpkg.title (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Softpkg.pkgtype (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Softpkg.abstract (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Softpkg.licenseURL (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.SoftpkgProperties '(Softpkg.Softpkg.name
```

```
   ,Softpkg.Softpkg.version ?
   ,Softpkg.Softpkg.title ?
   ,Softpkg.Softpkg.pkgtype ?
   ,Softpkg.Softpkg.abstract ?
   ,Softpkg.Softpkg.licenseURL ?)' >

<!ENTITY % Softpkg.SoftpkgCompositions '(Softpkg.Softpkg.implementation*
   ,Softpkg.Softpkg.IDL*
   ,Softpkg.Softpkg.propertyFile*
   ,Softpkg.Softpkg.dependency*
   ,Softpkg.Softpkg.descriptor*
   ,Softpkg.Softpkg.extension*
   ,Softpkg.Softpkg.author*)' >

<!ELEMENT Softpkg.Softpkg ( %Softpkg.SoftpkgProperties;
       ,  %Softpkg.SoftpkgCompositions; )?>

<!ATTLIST Softpkg.Softpkg %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Softpkg.Implementation                -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Softpkg.Implementation.UUID (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Implementation.type (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Implementation.abstract (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Implementation.processor (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Implementation.threadSafety EMPTY>
<!ATTLIST Softpkg.Implementation.threadSafety %Softpkg.ThreadSafetyKind;>

<!ELEMENT Softpkg.Implementation.humanLanguage (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.ImplementationProperties '(Softpkg.Implementation.UUID
   ,Softpkg.Implementation.type ?
   ,Softpkg.Implementation.abstract ?
   ,Softpkg.Implementation.processor *
   ,Softpkg.Implementation.threadSafety
   ,Softpkg.Implementation.humanLanguage ?)' >

<!ENTITY % Softpkg.ImplementationCompositions '(Softpkg.Implementation.OS
   ,Softpkg.Implementation.programmingLanguage
   ,Softpkg.Implementation.code?
   ,Softpkg.Implementation.compiler?
   ,Softpkg.Implementation.dependency*
   ,Softpkg.Implementation.extension*
   ,Softpkg.Implementation.descriptor*
   ,Softpkg.Implementation.propertyFile*
   ,Softpkg.Implementation.runTime*)' >
```

```
<!ELEMENT Softpkg.Implementation ( %Softpkg.ImplementationProperties;
       ,  %Softpkg.ImplementationCompositions; )?>

<!ATTLIST Softpkg.Implementation %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.IDL                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.IDL.id (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.IDLProperties '(Softpkg.IDL.id  )' >

<!ENTITY % Softpkg.IDLCompositions '(Softpkg.IDL.link?
   ,Softpkg.IDL.fileInArchive?
   ,Softpkg.IDL.repository?)' >

<!ELEMENT Softpkg.IDL ( %Softpkg.IDLProperties;
       ,  %Softpkg.IDLCompositions; )?>

<!ATTLIST Softpkg.IDL %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.PropertyFile                 -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.PropertyFile.implementation (Softpkg.Implementation) >

<!ENTITY % Softpkg.PropertyFileAssociations
'(Softpkg.PropertyFile.implementation?)' >

<!ENTITY % Softpkg.PropertyFileCompositions '(Softpkg.PropertyFile.link?
   ,Softpkg.PropertyFile.fileInArchive?)' >

<!ELEMENT Softpkg.PropertyFile ((XMI.extension* ,
%Softpkg.PropertyFileAssociations; )
       ,  %Softpkg.PropertyFileCompositions; )?>

<!ATTLIST Softpkg.PropertyFile %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.Dependency                   -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.Dependency.name (#PCDATA|XMI.reference)*>
```

```
<!ELEMENT Softpkg.Dependency.type (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Dependency.version (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Dependency.action EMPTY>
<!ATTLIST Softpkg.Dependency.action %Softpkg.ActionKind;>

<!ELEMENT Softpkg.Dependency.package (Softpkg.Softpkg)*>

<!ENTITY % Softpkg.DependencyProperties '(Softpkg.Dependency.name ?
   ,Softpkg.Dependency.type ?
   ,Softpkg.Dependency.version ?
   ,Softpkg.Dependency.action  )' >

<!ENTITY % Softpkg.DependencyAssociations '(Softpkg.Dependency.package*)'
>

<!ENTITY % Softpkg.DependencyCompositions '(Softpkg.Dependency.codeBase*
   ,Softpkg.Dependency.fileInArchive*
   ,Softpkg.Dependency.localFile*)' >

<!ELEMENT Softpkg.Dependency ( %Softpkg.DependencyProperties;
       ,(XMI.extension* ,   %Softpkg.DependencyAssociations; )
       ,  %Softpkg.DependencyCompositions; )?>

<!ATTLIST Softpkg.Dependency %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.Descriptor                  -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.Descriptor.type (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.DescriptorProperties '(Softpkg.Descriptor.type  )' >

<!ENTITY % Softpkg.DescriptorCompositions '(Softpkg.Descriptor.link?
   ,Softpkg.Descriptor.fileInArchive?)' >

<!ELEMENT Softpkg.Descriptor ( %Softpkg.DescriptorProperties;
       ,  %Softpkg.DescriptorCompositions; )?>

<!ATTLIST Softpkg.Descriptor %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.Author                      -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.Author.name (#PCDATA|XMI.reference)*>
```

```
<!ELEMENT Softpkg.Author.Company (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Author.WebPage (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.AuthorProperties '(Softpkg.Author.name ?
   ,Softpkg.Author.Company ?
   ,Softpkg.Author.WebPage ?)' >

<!ELEMENT Softpkg.Author ( %Softpkg.AuthorProperties; )?>

<!ATTLIST Softpkg.Author %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.OS                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.OS.name (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.OS.version (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.OSProperties '(Softpkg.OS.name
   ,Softpkg.OS.version ?)' >

<!ELEMENT Softpkg.OS ( %Softpkg.OSProperties; )?>

<!ATTLIST Softpkg.OS %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.ProgrammingLanguage         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.ProgrammingLanguage.name (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.ProgrammingLanguage.version (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.ProgrammingLanguageProperties
'(Softpkg.ProgrammingLanguage.name
   ,Softpkg.ProgrammingLanguage.version ?)' >

<!ELEMENT Softpkg.ProgrammingLanguage (
%Softpkg.ProgrammingLanguageProperties; )?>

<!ATTLIST Softpkg.ProgrammingLanguage %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.Code                        -->
```

```
<!--                                                                    -->
<!-- ______________________________________________________________ -->

<!ELEMENT Softpkg.Code.type (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.CodeProperties '(Softpkg.Code.type ?)' >

<!ENTITY % Softpkg.CodeCompositions '(Softpkg.Code.codeBase?
   ,Softpkg.Code.fileInArchive?
   ,Softpkg.Code.link?
   ,Softpkg.Code.entryPoint?)' >

<!ELEMENT Softpkg.Code ( %Softpkg.CodeProperties;
       ,  %Softpkg.CodeCompositions; )?>

<!ATTLIST Softpkg.Code %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: Softpkg.Compiler                      -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT Softpkg.Compiler.name (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.Compiler.version (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.CompilerProperties '(Softpkg.Compiler.name
   ,Softpkg.Compiler.version ?)' >

<!ELEMENT Softpkg.Compiler ( %Softpkg.CompilerProperties; )?>

<!ATTLIST Softpkg.Compiler %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________________ -->
<!--                                                                -->
<!-- METAMODEL CLASS: Softpkg.RunTime                       -->
<!--                                                                -->
<!-- ______________________________________________________________ -->

<!ELEMENT Softpkg.RunTime.name (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.RunTime.version (#PCDATA|XMI.reference)*>

<!ELEMENT Softpkg.RunTime.impelementation (Softpkg.Implementation) >

<!ENTITY % Softpkg.RunTimeProperties '(Softpkg.RunTime.name
   ,Softpkg.RunTime.version  )' >

<!ENTITY % Softpkg.RunTimeAssociations
'(Softpkg.RunTime.impelementation?)' >

<!ELEMENT Softpkg.RunTime ( %Softpkg.RunTimeProperties;
```

```
       ,(XMI.extension* ,   %Softpkg.RunTimeAssociations; ) )?>

<!ATTLIST Softpkg.RunTime %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Softpkg.EntryPoint                  -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Softpkg.EntryPoint.name (#PCDATA|XMI.reference)*>

<!ENTITY % Softpkg.EntryPointProperties '(Softpkg.EntryPoint.name  )' >

<!ELEMENT Softpkg.EntryPoint ( %Softpkg.EntryPointProperties; )?>

<!ATTLIST Softpkg.EntryPoint %XMI.element.att; %XMI.link.att; >

<!ELEMENT Softpkg ((Softpkg.Softpkg
  |Softpkg.Implementation
  |Softpkg.IDL
  |Softpkg.PropertyFile
  |Softpkg.Dependency
  |Softpkg.Descriptor
  |Softpkg.Author
  |Softpkg.OS
  |Softpkg.ProgrammingLanguage
  |Softpkg.Code
  |Softpkg.Compiler
  |Softpkg.RunTime
  |Softpkg.EntryPoint)*)>
<!ATTLIST Softpkg %XMI.element.att; %XMI.link.att;>



<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE: Component                         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->


<!-- *****    Component.Container_EventPolicy   *******  -->

<!ELEMENT Component.Component.eventPolicy ( Component.EventPolicy)? >


<!-- *****    Component.Container_Extension   *******  -->

<!ELEMENT Component.Component.extension ( PDGeneral.Extension)* >


<!-- *****    Component.Container_Servant   *******  -->
```

```
<!ELEMENT Component.Component.servant ( Component.Servant)? >


<!-- *****    Component.Container_SecuritySpecifier   *******  -->

<!ELEMENT Component.Component.securitySpecifier (
Component.SecuritySpecifier)? >


<!-- *****    Component.Component_ExtendedPOAPolicy   *******  -->

<!ELEMENT Component.Component.extPOAPolicy (
Component.ExtendedPOAPolicy)? >


<!-- *****    Component.Component_Interface   *******  -->

<!ELEMENT Component.Component.supports ( Component.Interface)* >


<!-- *****    Component.Component_Port   *******  -->

<!ELEMENT Component.Component.port ( Component.Port
  |Component.InterfacePort
  |Component.EventPort)+ >


<!-- *****    Component.EventPort_ValueType   *******  -->

<!ELEMENT Component.EventPort.valueType ( Component.ValueType)* >


<!-- *****    Component.InterfacePort_Interface   *******  -->

<!ELEMENT Component.InterfacePort.interface ( Component.Interface)* >


<!-- *****    Component.Component_Persistence   *******  -->

<!ELEMENT Component.Component.persistence ( Component.Persistence)? >


<!-- *****    Component.Component_POAPolicy   *******  -->

<!ELEMENT Component.Component.poaPolicy ( Component.POAPolicy)? >


<!-- *****    Component.Component_PersistentStoreInfo   *******  -->

<!ELEMENT Component.Component.persistentStoreInfo (
Component.PersistentStoreInfo)? >


<!-- *****    Component.Component_Repository   *******  -->
```

```
<!ELEMENT Component.Component.repository ( PDGeneral.Repository)? >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.Component                 -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Component.Component.name (#PCDATA|XMI.reference)*>

<!ELEMENT Component.Component.repositoryId (#PCDATA|XMI.reference)*>

<!ELEMENT Component.Component.corbaVersion (#PCDATA|XMI.reference)*>

<!ELEMENT Component.Component.componentKind EMPTY>
<!ATTLIST Component.Component.componentKind %Component.ComponentKind;>

<!ELEMENT Component.Component.transactionSupport EMPTY>
<!ATTLIST Component.Component.transactionSupport
%Component.TransactionSupportKind;>

<!ELEMENT Component.Component.securityCredentialKind EMPTY>
<!ATTLIST Component.Component.securityCredentialKind
%Component.SecurityCredentialKind;>

<!ELEMENT Component.Component.containerThreading EMPTY>
<!ATTLIST Component.Component.containerThreading
%Component.ContainerThreadingKind;>

<!ELEMENT Component.Component.configurationComplete EMPTY>
<!ATTLIST Component.Component.configurationComplete
        XMI.value ( true | false ) #REQUIRED>

<!ELEMENT Component.Component.supportedInterfaceRepId
(#PCDATA|XMI.reference)*>

<!ELEMENT Component.Component.base (Component.Component)?>

<!ENTITY % Component.ComponentProperties '(Component.Component.name
   ,Component.Component.repositoryId
   ,Component.Component.corbaVersion
   ,Component.Component.componentKind
   ,Component.Component.transactionSupport ?
   ,Component.Component.securityCredentialKind ?
   ,Component.Component.containerThreading
   ,Component.Component.configurationComplete
   ,Component.Component.supportedInterfaceRepId ?)' >

<!ENTITY % Component.ComponentAssociations '(Component.Component.base?)'
>

<!ENTITY % Component.ComponentCompositions
'(Component.Component.eventPolicy?
   ,Component.Component.extension*
```

```
   ,Component.Component.servant?
   ,Component.Component.securitySpecifier?
   ,Component.Component.extPOAPolicy?
   ,Component.Component.supports*
   ,Component.Component.port+
   ,Component.Component.persistence?
   ,Component.Component.poaPolicy?
   ,Component.Component.persistentStoreInfo?
   ,Component.Component.repository?)' >

<!ELEMENT Component.Component ( %Component.ComponentProperties;
       ,(XMI.extension* ,   %Component.ComponentAssociations; )
       ,  %Component.ComponentCompositions; )?>

<!ATTLIST Component.Component %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Component.EventPolicy                 -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Component.EventPolicy.emit EMPTY>
<!ATTLIST Component.EventPolicy.emit %Component.EventPolicyKind;>

<!ELEMENT Component.EventPolicy.consume EMPTY>
<!ATTLIST Component.EventPolicy.consume %Component.EventPolicyKind;>

<!ENTITY % Component.EventPolicyProperties '(Component.EventPolicy.emit ?
   ,Component.EventPolicy.consume ?)' >

<!ELEMENT Component.EventPolicy ( %Component.EventPolicyProperties; )?>

<!ATTLIST Component.EventPolicy %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Component.Servant                     -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Component.Servant.LifeTime EMPTY>
<!ATTLIST Component.Servant.LifeTime %Component.LifeTimeKind;>

<!ENTITY % Component.ServantProperties '(Component.Servant.LifeTime  )' >

<!ELEMENT Component.Servant ( %Component.ServantProperties; )?>

<!ATTLIST Component.Servant %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
```

```
<!-- METAMODEL CLASS: Component.SecuritySpecifier           -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ELEMENT Component.SecuritySpecifier.userId (#PCDATA|XMI.reference)*>

<!ENTITY % Component.SecuritySpecifierProperties
'(Component.SecuritySpecifier.userId  )' >

<!ELEMENT Component.SecuritySpecifier (
%Component.SecuritySpecifierProperties; )?>

<!ATTLIST Component.SecuritySpecifier %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________ -->
<!--                                                        -->
<!-- METAMODEL CLASS: Component.ExtendedPOAPolicy           -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ELEMENT Component.ExtendedPOAPolicy.name (#PCDATA|XMI.reference)*>

<!ELEMENT Component.ExtendedPOAPolicy.value (#PCDATA|XMI.reference)*>

<!ENTITY % Component.ExtendedPOAPolicyProperties
'(Component.ExtendedPOAPolicy.name
   ,Component.ExtendedPOAPolicy.value  )' >

<!ELEMENT Component.ExtendedPOAPolicy (
%Component.ExtendedPOAPolicyProperties; )?>

<!ATTLIST Component.ExtendedPOAPolicy %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________ -->
<!--                                                        -->
<!-- METAMODEL CLASS: Component.ObjectType                  -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ELEMENT Component.ObjectType.name (#PCDATA|XMI.reference)*>

<!ELEMENT Component.ObjectType.repositoryId (#PCDATA|XMI.reference)*>

<!ENTITY % Component.ObjectTypeProperties '(Component.ObjectType.name
   ,Component.ObjectType.repositoryId  )' >

<!ELEMENT Component.ObjectType ( %Component.ObjectTypeProperties; )?>

<!ATTLIST Component.ObjectType %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________ -->
<!--                                                        -->
```

```
<!-- METAMODEL CLASS: Component.Interface                    -->
<!--                                                         -->
<!-- _______________________________________________________ -->

<!ELEMENT Component.Interface.base (Component.Interface)*>

<!ENTITY % Component.InterfaceProperties
'(%Component.ObjectTypeProperties;)' >

<!ENTITY % Component.InterfaceAssociations '(Component.Interface.base*)'
>

<!ELEMENT Component.Interface ( %Component.InterfaceProperties;
       ,(XMI.extension* ,   %Component.InterfaceAssociations; ) )?>

<!ATTLIST Component.Interface %XMI.element.att; %XMI.link.att; >


<!-- _______________________________________________________ -->
<!--                                                         -->
<!-- METAMODEL CLASS: Component.Port                         -->
<!--                                                         -->
<!-- _______________________________________________________ -->

<!ELEMENT Component.Port.name (#PCDATA|XMI.reference)*>

<!ENTITY % Component.PortProperties '(Component.Port.name  )' >

<!ELEMENT Component.Port ( %Component.PortProperties; )?>

<!ATTLIST Component.Port %XMI.element.att; %XMI.link.att; >


<!-- _______________________________________________________ -->
<!--                                                         -->
<!-- METAMODEL CLASS: Component.Persistence                  -->
<!--                                                         -->
<!-- _______________________________________________________ -->

<!ELEMENT Component.Persistence.responsibility EMPTY>
<!ATTLIST Component.Persistence.responsibility
%Component.PersistenceResponsibilityKind;>

<!ELEMENT Component.Persistence.usePSS EMPTY>
<!ATTLIST Component.Persistence.usePSS
        XMI.value ( true | false ) #REQUIRED>

<!ENTITY % Component.PersistenceProperties
'(Component.Persistence.responsibility
   ,Component.Persistence.usePSS  )' >

<!ELEMENT Component.Persistence ( %Component.PersistenceProperties; )?>

<!ATTLIST Component.Persistence %XMI.element.att; %XMI.link.att; >
```

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.POAPolicy                         -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Component.POAPolicy.idAssignment EMPTY>
<!ATTLIST Component.POAPolicy.idAssignment
%Component.POAIdAssignmentPolicy;>

<!ELEMENT Component.POAPolicy.idUniqueness EMPTY>
<!ATTLIST Component.POAPolicy.idUniqueness
%Component.POAIdUniquenessPolicy;>

<!ELEMENT Component.POAPolicy.implicitActivation EMPTY>
<!ATTLIST Component.POAPolicy.implicitActivation
%Component.POAImplicitActivationPolicy;>

<!ELEMENT Component.POAPolicy.lifeSpan EMPTY>
<!ATTLIST Component.POAPolicy.lifeSpan %Component.POALifeSpanPolicy;>

<!ELEMENT Component.POAPolicy.requestProcessing EMPTY>
<!ATTLIST Component.POAPolicy.requestProcessing
%Component.POARequestProcessingPolicy;>

<!ELEMENT Component.POAPolicy.servantRetention EMPTY>
<!ATTLIST Component.POAPolicy.servantRetention
%Component.POAServantRetentionPolicy;>

<!ELEMENT Component.POAPolicy.thread EMPTY>
<!ATTLIST Component.POAPolicy.thread %Component.POAThreadPolicy;>

<!ENTITY % Component.POAPolicyProperties
'(Component.POAPolicy.idAssignment
   ,Component.POAPolicy.idUniqueness
   ,Component.POAPolicy.implicitActivation
   ,Component.POAPolicy.lifeSpan
   ,Component.POAPolicy.requestProcessing
   ,Component.POAPolicy.servantRetention
   ,Component.POAPolicy.thread  )' >

<!ELEMENT Component.POAPolicy ( %Component.POAPolicyProperties; )?>

<!ATTLIST Component.POAPolicy %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.PersistentStoreInfo               -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Component.PersistentStoreInfo.implementation
(#PCDATA|XMI.reference)*>
```

```
<!ELEMENT Component.PersistentStoreInfo.dataStoreName
(#PCDATA|XMI.reference)*>

<!ELEMENT Component.PersistentStoreInfo.dataStoreId
(#PCDATA|XMI.reference)*>

<!ENTITY % Component.PersistentStoreInfoProperties
'(Component.PersistentStoreInfo.implementation
   ,Component.PersistentStoreInfo.dataStoreName
   ,Component.PersistentStoreInfo.dataStoreId  )' >

<!ELEMENT Component.PersistentStoreInfo (
%Component.PersistentStoreInfoProperties; )?>

<!ATTLIST Component.PersistentStoreInfo %XMI.element.att; %XMI.link.att;
>


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.InterfacePort              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Component.InterfacePort.kind EMPTY>
<!ATTLIST Component.InterfacePort.kind %Component.InterfacePortKind;>

<!ENTITY % Component.InterfacePortProperties '(%Component.PortProperties;
   ,Component.InterfacePort.kind  )' >

<!ENTITY % Component.InterfacePortCompositions
'(Component.InterfacePort.interface*)' >

<!ELEMENT Component.InterfacePort ( %Component.InterfacePortProperties;
       ,  %Component.InterfacePortCompositions; )?>

<!ATTLIST Component.InterfacePort %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.EventPort                  -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Component.EventPort.kind EMPTY>
<!ATTLIST Component.EventPort.kind %Component.EventPortKind;>

<!ENTITY % Component.EventPortProperties '(%Component.PortProperties;
   ,Component.EventPort.kind  )' >

<!ENTITY % Component.EventPortCompositions
'(Component.EventPort.valueType*)' >
```

```
<!ELEMENT Component.EventPort ( %Component.EventPortProperties;
       ,  %Component.EventPortCompositions; )?>

<!ATTLIST Component.EventPort %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Component.ValueType                 -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Component.ValueTypeProperties
'(%Component.ObjectTypeProperties;)' >

<!ELEMENT Component.ValueType ( %Component.ValueTypeProperties; )?>

<!ATTLIST Component.ValueType %XMI.element.att; %XMI.link.att; >

<!ELEMENT Component ((Component.Component
  |Component.EventPolicy
  |Component.Servant
  |Component.SecuritySpecifier
  |Component.ExtendedPOAPolicy
  |Component.Interface
  |Component.Port
  |Component.Persistence
  |Component.POAPolicy
  |Component.PersistentStoreInfo
  |Component.ObjectType
  |Component.InterfacePort
  |Component.EventPort
  |Component.ValueType)*)>
<!ATTLIST Component %XMI.element.att; %XMI.link.att;>



<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE: Assembly                          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->


<!-- *****    Assembly.Assembly_Component   *******  -->

<!ELEMENT Assembly.Assembly.componentFile ( Assembly.ComponentFile)+ >


<!-- *****    Assembly.Assembly_Partitioning   *******  -->

<!ELEMENT Assembly.Assembly.partitioning ( Assembly.Partitioning)* >


<!-- *****    Assembly.Assembly_Connection   *******  -->
```

```
<!ELEMENT Assembly.Assembly.connections ( Assembly.Connections)* >


<!-- *****    Assembly.Assembly_Extension   *******  -->

<!ELEMENT Assembly.Assembly.extension ( PDGeneral.Extension)* >


<!-- *****    Assembly.ComponentFile_CodeBase   *******  -->

<!ELEMENT Assembly.ComponentFile.codeBase ( PDGeneral.CodeBase)? >


<!-- *****    Assembly.ComponentFile_Link   *******  -->

<!ELEMENT Assembly.ComponentFile.link ( PDGeneral.Link)? >


<!-- *****    Assembly.ComponentFile_FileInArchive   *******  -->

<!ELEMENT Assembly.ComponentFile.fileInArchive (
PDGeneral.FileInArchive)? >


<!-- *****    Assembly.Partitioning_ComponentPlacement   *******  -->

<!ELEMENT Assembly.Partitioning.componentPlacement (
Assembly.ComponentPlacement)* >


<!-- *****    Assembly.A_partitioning_collocation   *******  -->

<!ELEMENT Assembly.Partitioning.Collocation ( Assembly.Collocation
  |Assembly.ProcessCollocation
  |Assembly.HostCollocation)* >


<!-- *****    Assembly.Partitioning_ProcessCollocation   *******  -->

<!ELEMENT Assembly.Partitioning.process ( Assembly.ProcessCollocation)* >


<!-- *****    Assembly.Partitioning_Extension   *******  -->

<!ELEMENT Assembly.Partitioning.extension ( PDGeneral.Extension)* >


<!-- *****    Assembly.Collocation_ComponentPlacement   *******  -->

<!ELEMENT Assembly.Collocation.placement ( Assembly.ComponentPlacement)+
>


<!-- *****    Assembly.Collocation_Extension   *******  -->
```

```
<!ELEMENT Assembly.Collocation.extension ( PDGeneral.Extension)* >


<!-- *****    Assembly.HostCollocation_ProcessCollocation   *******  -->

<!ELEMENT Assembly.HostCollocation.process (
Assembly.ProcessCollocation)+ >


<!-- *****    Assembly.Partitioning_HostCollocation   *******  -->

<!ELEMENT Assembly.Partitioning.host ( Assembly.HostCollocation)* >


<!-- *****    Assembly.PropertiesFile_CodeBase   *******  -->

<!ELEMENT Assembly.PropertiesFile.codeBase ( PDGeneral.CodeBase)? >


<!-- *****    Assembly.PropertiesFile_FileInArchive   *******  -->

<!ELEMENT Assembly.PropertiesFile.fileInArchive (
PDGeneral.FileInArchive)? >


<!-- *****    Assembly.ComponentPlacement_PropertiesFile   *******  -->

<!ELEMENT Assembly.ComponentPlacement.propertiesFile (
Assembly.PropertiesFile)? >


<!-- *****    Assembly.ComponentPlacement_Extension   *******  -->

<!ELEMENT Assembly.ComponentPlacement.extension ( PDGeneral.Extension)* >


<!-- *****    Assembly.ComponentPlacement_TraderProperties   *******  -->

<!ELEMENT Assembly.ComponentPlacement.traderProperties (
Assembly.TraderProperties)* >


<!-- *****    Assembly.Connections_Extension   *******  -->

<!ELEMENT Assembly.Connections.extension ( PDGeneral.Extension)* >


<!-- *****    Assembly.Connections_ConnectEvent   *******  -->

<!ELEMENT Assembly.Connections.connectEvent ( Assembly.ConnectEvent)* >


<!-- *****    Assembly.Connections_ConnectInterface   *******  -->
```

```
<!ELEMENT Assembly.Connections.connectInterface (
Assembly.ConnectInterface)* >


<!-- *****    Assembly.ConnectInterface_ProvidesInterface   *******  -->

<!ELEMENT Assembly.ConnectInterface.provides (
Assembly.ProvidesInterface)* >


<!-- *****    Assembly.ConnectInterface_UsesInterface   *******  -->

<!ELEMENT Assembly.ConnectInterface.uses ( Assembly.UsesInterface)* >


<!-- *****    Assembly.ConnectEvent_EmitsEvent   *******  -->

<!ELEMENT Assembly.ConnectEvent.emits ( Assembly.EmitsEvent)* >


<!-- *****    Assembly.ConnectEvent_ConsumesEvent   *******  -->

<!ELEMENT Assembly.ConnectEvent.consumes ( Assembly.ConsumesEvent)* >


<!-- *****    Assembly.ConnectEvent_PublishesEvent   *******  -->

<!ELEMENT Assembly.ConnectEvent.publishes ( Assembly.PublishesEvent)* >


<!-- *****    Assembly.ComponentFile_PropertiesFile   *******  -->

<!ELEMENT Assembly.ComponentFile.propertiesFile (
Assembly.PropertiesFile)? >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.Assembly                       -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.AssemblyCompositions
'(Assembly.Assembly.componentFile+
   ,Assembly.Assembly.partitioning*
   ,Assembly.Assembly.connections*
   ,Assembly.Assembly.extension*)' >

<!ELEMENT Assembly.Assembly (  %Assembly.AssemblyCompositions; )?>

<!ATTLIST Assembly.Assembly %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
```

```
<!-- METAMODEL CLASS: Assembly.ComponentFile                -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ELEMENT Assembly.ComponentFile.id (#PCDATA|XMI.reference)*>

<!ENTITY % Assembly.ComponentFileProperties '(Assembly.ComponentFile.id
)' >

<!ENTITY % Assembly.ComponentFileCompositions
'(Assembly.ComponentFile.codeBase?
   ,Assembly.ComponentFile.link?
   ,Assembly.ComponentFile.fileInArchive?
   ,Assembly.ComponentFile.propertiesFile?)' >

<!ELEMENT Assembly.ComponentFile ( %Assembly.ComponentFileProperties;
       ,  %Assembly.ComponentFileCompositions; )?>

<!ATTLIST Assembly.ComponentFile %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________ -->
<!--                                                        -->
<!-- METAMODEL CLASS: Assembly.Partitioning                 -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ENTITY % Assembly.PartitioningCompositions
'(Assembly.Partitioning.componentPlacement*
   ,Assembly.Partitioning.Collocation*
   ,Assembly.Partitioning.process*
   ,Assembly.Partitioning.extension*
   ,Assembly.Partitioning.host*)' >

<!ELEMENT Assembly.Partitioning (  %Assembly.PartitioningCompositions;
)?>

<!ATTLIST Assembly.Partitioning %XMI.element.att; %XMI.link.att; >


<!-- ______________________________________________________ -->
<!--                                                        -->
<!-- METAMODEL CLASS: Assembly.Connections                  -->
<!--                                                        -->
<!-- ______________________________________________________ -->

<!ENTITY % Assembly.ConnectionsCompositions
'(Assembly.Connections.extension*
   ,Assembly.Connections.connectEvent*
   ,Assembly.Connections.connectInterface*)' >

<!ELEMENT Assembly.Connections (  %Assembly.ConnectionsCompositions; )?>

<!ATTLIST Assembly.Connections %XMI.element.att; %XMI.link.att; >
```

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.PropertiesFile              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.PropertiesFileCompositions
'(Assembly.PropertiesFile.codeBase?
   ,Assembly.PropertiesFile.fileInArchive?)' >

<!ELEMENT Assembly.PropertiesFile (
%Assembly.PropertiesFileCompositions; )?>

<!ATTLIST Assembly.PropertiesFile %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.ComponentPlacement          -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Assembly.ComponentPlacement.id (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentPlacement.usageName (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentPlacement.objectReference
(#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentPlacement.registerWithNaming
(#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentPlacement.cardinality
(#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentPlacement.componentFile
(Assembly.ComponentFile)?>

<!ELEMENT Assembly.ComponentPlacement.implementation
(Softpkg.Implementation)?>

<!ENTITY % Assembly.ComponentPlacementProperties
'(Assembly.ComponentPlacement.id
   ,Assembly.ComponentPlacement.usageName ?
   ,Assembly.ComponentPlacement.objectReference ?
   ,Assembly.ComponentPlacement.registerWithNaming *
   ,Assembly.ComponentPlacement.cardinality  )' >

<!ENTITY % Assembly.ComponentPlacementAssociations
'(Assembly.ComponentPlacement.componentFile?
   ,Assembly.ComponentPlacement.implementation?)' >

<!ENTITY % Assembly.ComponentPlacementCompositions
'(Assembly.ComponentPlacement.propertiesFile?
```

```
   ,Assembly.ComponentPlacement.extension*
   ,Assembly.ComponentPlacement.traderProperties*)' >

<!ELEMENT Assembly.ComponentPlacement (
%Assembly.ComponentPlacementProperties;
       ,(XMI.extension* ,   %Assembly.ComponentPlacementAssociations; )
       ,  %Assembly.ComponentPlacementCompositions; )?>

<!ATTLIST Assembly.ComponentPlacement %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Assembly.Collocation                  -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Assembly.Collocation.id (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.Collocation.usageName (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.Collocation.implementationType
(#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.Collocation.cardinality (#PCDATA|XMI.reference)*>

<!ENTITY % Assembly.CollocationProperties '(Assembly.Collocation.id
   ,Assembly.Collocation.usageName
   ,Assembly.Collocation.implementationType ?
   ,Assembly.Collocation.cardinality  )' >

<!ENTITY % Assembly.CollocationCompositions
'(Assembly.Collocation.placement+
   ,Assembly.Collocation.extension*)' >

<!ELEMENT Assembly.Collocation ( %Assembly.CollocationProperties;
       ,  %Assembly.CollocationCompositions; )?>

<!ATTLIST Assembly.Collocation %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Assembly.ProcessCollocation           -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Assembly.ProcessCollocation.host (Assembly.HostCollocation) >

<!ENTITY % Assembly.ProcessCollocationProperties
'(%Assembly.CollocationProperties;)' >

<!ENTITY % Assembly.ProcessCollocationAssociations
'(Assembly.ProcessCollocation.host?)' >
```

```
<!ENTITY % Assembly.ProcessCollocationCompositions
'(%Assembly.CollocationCompositions;)' >

<!ELEMENT Assembly.ProcessCollocation (
%Assembly.ProcessCollocationProperties;
       ,(XMI.extension* ,   %Assembly.ProcessCollocationAssociations; )
       ,  %Assembly.ProcessCollocationCompositions; )?>

<!ATTLIST Assembly.ProcessCollocation %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.HostCollocation              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.HostCollocationProperties
'(%Assembly.CollocationProperties;)' >

<!ENTITY % Assembly.HostCollocationCompositions
'(%Assembly.CollocationCompositions;
   ,Assembly.HostCollocation.process+)' >

<!ELEMENT Assembly.HostCollocation ( %Assembly.HostCollocationProperties;
       ,  %Assembly.HostCollocationCompositions; )?>

<!ATTLIST Assembly.HostCollocation %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.Connect                      -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Assembly.Connect.id (#PCDATA|XMI.reference)*>

<!ENTITY % Assembly.ConnectProperties '(Assembly.Connect.id  )' >

<!ELEMENT Assembly.Connect ( %Assembly.ConnectProperties; )?>

<!ATTLIST Assembly.Connect %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.ConnectEvent                 -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.ConnectEventProperties
'(%Assembly.ConnectProperties;)' >
```

```
<!ENTITY % Assembly.ConnectEventCompositions
'(Assembly.ConnectEvent.emits*
   ,Assembly.ConnectEvent.consumes*
   ,Assembly.ConnectEvent.publishes*)' >

<!ELEMENT Assembly.ConnectEvent ( %Assembly.ConnectEventProperties;
       ,  %Assembly.ConnectEventCompositions; )?>

<!ATTLIST Assembly.ConnectEvent %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.ConnectInterface             -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.ConnectInterfaceProperties
'(%Assembly.ConnectProperties;)' >

<!ENTITY % Assembly.ConnectInterfaceCompositions
'(Assembly.ConnectInterface.provides*
   ,Assembly.ConnectInterface.uses*)' >

<!ELEMENT Assembly.ConnectInterface (
%Assembly.ConnectInterfaceProperties;
       ,  %Assembly.ConnectInterfaceCompositions; )?>

<!ATTLIST Assembly.ConnectInterface %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.TraderProperties             -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Assembly.TraderProperties.name (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.TraderProperties.value (#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.TraderProperties.placement
(Assembly.ComponentPlacement) >

<!ENTITY % Assembly.TraderPropertiesProperties
'(Assembly.TraderProperties.name
   ,Assembly.TraderProperties.value  )' >

<!ENTITY % Assembly.TraderPropertiesAssociations
'(Assembly.TraderProperties.placement?)' >

<!ELEMENT Assembly.TraderProperties (
%Assembly.TraderPropertiesProperties;
       ,(XMI.extension* ,   %Assembly.TraderPropertiesAssociations; ) )?>
```

```
<!ATTLIST Assembly.TraderProperties %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.ComponentElementReference    -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Assembly.ComponentElementReference.elementIdentifier
(#PCDATA|XMI.reference)*>

<!ELEMENT Assembly.ComponentElementReference.component
(Component.Component) >

<!ENTITY % Assembly.ComponentElementReferenceProperties
'(Assembly.ComponentElementReference.elementIdentifier  )' >

<!ENTITY % Assembly.ComponentElementReferenceAssociations
'(Assembly.ComponentElementReference.component )' >

<!ELEMENT Assembly.ComponentElementReference (
%Assembly.ComponentElementReferenceProperties;
       ,(XMI.extension* ,
%Assembly.ComponentElementReferenceAssociations; ) )?>

<!ATTLIST Assembly.ComponentElementReference %XMI.element.att;
%XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.ProvidesInterface            -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % Assembly.ProvidesInterfaceProperties
'(%Assembly.ComponentElementReferenceProperties;)' >

<!ENTITY % Assembly.ProvidesInterfaceAssociations
'(%Assembly.ComponentElementReferenceAssociations;)' >

<!ELEMENT Assembly.ProvidesInterface (
%Assembly.ProvidesInterfaceProperties;
       ,(XMI.extension* ,   %Assembly.ProvidesInterfaceAssociations; ) )?>

<!ATTLIST Assembly.ProvidesInterface %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.UsesInterface                -->
<!--                                                              -->
<!-- ____________________________________________________________ -->
```

```
<!ENTITY % Assembly.UsesInterfaceProperties
'(%Assembly.ComponentElementReferenceProperties;)' >

<!ENTITY % Assembly.UsesInterfaceAssociations
'(%Assembly.ComponentElementReferenceAssociations;)' >

<!ELEMENT Assembly.UsesInterface ( %Assembly.UsesInterfaceProperties;
       ,(XMI.extension* ,   %Assembly.UsesInterfaceAssociations; ) )?>

<!ATTLIST Assembly.UsesInterface %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Assembly.EmitsEvent                    -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ENTITY % Assembly.EmitsEventProperties
'(%Assembly.ComponentElementReferenceProperties;)' >

<!ENTITY % Assembly.EmitsEventAssociations
'(%Assembly.ComponentElementReferenceAssociations;)' >

<!ELEMENT Assembly.EmitsEvent ( %Assembly.EmitsEventProperties;
       ,(XMI.extension* ,   %Assembly.EmitsEventAssociations; ) )?>

<!ATTLIST Assembly.EmitsEvent %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Assembly.ConsumesEvent                 -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ENTITY % Assembly.ConsumesEventProperties
'(%Assembly.ComponentElementReferenceProperties;)' >

<!ENTITY % Assembly.ConsumesEventAssociations
'(%Assembly.ComponentElementReferenceAssociations;)' >

<!ELEMENT Assembly.ConsumesEvent ( %Assembly.ConsumesEventProperties;
       ,(XMI.extension* ,   %Assembly.ConsumesEventAssociations; ) )?>

<!ATTLIST Assembly.ConsumesEvent %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________________ -->
<!--                                                                      -->
<!-- METAMODEL CLASS: Assembly.PublishesEvent                -->
<!--                                                                      -->
<!-- ____________________________________________________________________ -->

<!ELEMENT Assembly.PublishesEvent.connect (Assembly.ConnectEvent) >
```

```
<!ENTITY % Assembly.PublishesEventAssociations
'(Assembly.PublishesEvent.connect?)' >

<!ELEMENT Assembly.PublishesEvent ((XMI.extension* ,
%Assembly.PublishesEventAssociations; ) )?>

<!ATTLIST Assembly.PublishesEvent %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: Assembly.Connection                 -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT Assembly.Connection (EMPTY )>

<!ATTLIST Assembly.Connection %XMI.element.att; %XMI.link.att; >

<!ELEMENT Assembly ((Assembly.Assembly
  |Assembly.ComponentFile
  |Assembly.Partitioning
  |Assembly.Connections
  |Assembly.PropertiesFile
  |Assembly.ComponentPlacement
  |Assembly.Collocation
  |Assembly.ProcessCollocation
  |Assembly.HostCollocation
  |Assembly.ConnectEvent
  |Assembly.ConnectInterface
  |Assembly.TraderProperties
  |Assembly.Connect
  |Assembly.ProvidesInterface
  |Assembly.UsesInterface
  |Assembly.EmitsEvent
  |Assembly.ConsumesEvent
  |Assembly.PublishesEvent
  |Assembly.ComponentElementReference
  |Assembly.Connection)*)>
<!ATTLIST Assembly %XMI.element.att; %XMI.link.att;>



<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL PACKAGE: PropertySet                       -->
<!--                                                              -->
<!-- ____________________________________________________________ -->


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PropertySet.Complex                 -->
<!--                                                              -->
```

```
<!-- ____________________________________________________________ -->

<!ELEMENT PropertySet.Complex.name (#PCDATA|XMI.reference)*>

<!ELEMENT PropertySet.Complex.type (#PCDATA|XMI.reference)*>

<!ELEMENT PropertySet.Complex.description (#PCDATA|XMI.reference)*>

<!ENTITY % PropertySet.ComplexProperties '(PropertySet.Complex.name ?
   ,PropertySet.Complex.type
   ,PropertySet.Complex.description  )' >

<!ELEMENT PropertySet.Complex ( %PropertySet.ComplexProperties; )?>

<!ATTLIST PropertySet.Complex %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PropertySet.Struct                  -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % PropertySet.StructProperties
'(%PropertySet.ComplexProperties;)' >

<!ELEMENT PropertySet.Struct ( %PropertySet.StructProperties; )?>

<!ATTLIST PropertySet.Struct %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PropertySet.Simple                  -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT PropertySet.Simple.type EMPTY>
<!ATTLIST PropertySet.Simple.type %PropertySet.SimpleType;>

<!ELEMENT PropertySet.Simple.defaultValue (#PCDATA|XMI.reference)*>

<!ELEMENT PropertySet.Simple.value (#PCDATA|XMI.reference)*>

<!ELEMENT PropertySet.Simple.description (#PCDATA|XMI.reference)*>

<!ELEMENT PropertySet.Simple.choice (#PCDATA|XMI.reference)*>

<!ENTITY % PropertySet.SimpleProperties '(PropertySet.Simple.type
   ,PropertySet.Simple.defaultValue ?
   ,PropertySet.Simple.value
   ,PropertySet.Simple.description ?
   ,PropertySet.Simple.choice *)' >

<!ELEMENT PropertySet.Simple ( %PropertySet.SimpleProperties; )?>
```

```
<!ATTLIST PropertySet.Simple %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PropertySet.Sequence                -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ENTITY % PropertySet.SequenceProperties
'(%PropertySet.ComplexProperties;)' >

<!ELEMENT PropertySet.Sequence ( %PropertySet.SequenceProperties; )?>

<!ATTLIST PropertySet.Sequence %XMI.element.att; %XMI.link.att; >


<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- METAMODEL CLASS: PropertySet.Properties              -->
<!--                                                              -->
<!-- ____________________________________________________________ -->

<!ELEMENT PropertySet.Properties.description (#PCDATA|XMI.reference)*>

<!ENTITY % PropertySet.PropertiesProperties
'(PropertySet.Properties.description ?)' >

<!ELEMENT PropertySet.Properties ( %PropertySet.PropertiesProperties; )?>

<!ATTLIST PropertySet.Properties %XMI.element.att; %XMI.link.att; >

<!ELEMENT PropertySet ((PropertySet.Struct
  |PropertySet.Simple
  |PropertySet.Sequence
  |PropertySet.Properties
  |PropertySet.Complex)*)>
<!ATTLIST PropertySet %XMI.element.att; %XMI.link.att;>
```

Related Work D

This chapter explores languages and programming idioms which support the expression of multiple interfaces and interface dependencies.

D.1 Polymorphism

A popular idiom in object-oriented programming is for an object to depend on an interface or an abstract base class. At runtime the object may receive a reference to the interface, which is dynamically bound to its implementation.

The programmer usually becomes aware of the dependency by examining method parameters or by reading comments or documentation. That is, there is no first class language constructs to highlight the interface dependencies.

D.2 Java Parameterized Type Proposals

Emerging proposals for parameterized types in Java have introduced interesting mechanisms for expressing type dependencies. Parameterized types express dependencies on one or more other types, to be determined when the template is *instantiated*.

D.2.1 Where Clauses

In the language Theta [Liskov 95] and in a recent proposal for parameterized types in Java [Myers 97], parameters to parameterized types are constrained by *where clauses*. Where clauses state explicitly which methods a parameter type must support in order to be used as a parameter to the parameterized type. This is a first class language construct for stating method dependencies. For example

For example, in proposed Java syntax:

```
interface Set [T]
    where T { boolean equals(T t); }
{ ... }
```

D.2.2 Constraining on Interface

Another proposal for parameterized types in Java [Agesen 97], allows type parameters to be constrained to support a particular interface. That is, only types which implement the given interface may be supplied as a type parameter to the parameterized type.

For example:

```
interface Equal<T> {
    boolean equal(T);
}

class Set<T implements Equal<T>>
{ ... }
```

This mechanism is analogous to this submission's specification of the **uses** statement for specifying a required interface.

D.3 JavaBeans

JavaBeans sidesteps the lack of first class language constructs to describe an object's interface dependencies. It uses a combination of naming conventions, introspection and external representation to describe certain types of interface dependencies.

The JavaBeans specification provides a clever mechanism for a class to express its runtime interface dependencies via a set of naming conventions and programming idioms, which they call "design patterns"[1]. An introspector looks for these naming conventions to determine what events a Bean generates and what kind of listener interfaces may register with it. This information is stored in a BeanInfo object. A BeanInfo may also be written by hand, circumventing the introspection process (and allowing deviation from prescribed naming conventions).

JavaBeans typically communicate with registered interfaces using events. The event, which is usually a data object that communicates information about something that happened, is transferred to the listening object via a method call.

1. An unfortunate choice of terms as JavaBeans design patterns are quite different from the design patterns as known to the design patterns community and expressed in the Design Patterns book [Gamma 95].

D.4 COM

In COM [Rogerson 97], the interfaces that a component provides are specified in the IDL specification of a component. A component is declared by a **coclass** declaration in an IDL file. A **coclass** declares the interfaces that it *provides* by listing each in the declaration. The interfaces that a component *uses* are specified as **source** interfaces. The **source** modifier indicates that the component is the source of calls to that interface. A component with **source** interfaces must also provide the **IConnectionPointContainer** interface. **IConnectionPointContainer** is used by clients to query an objects source interfaces and to register their client interfaces as *sinks* for the *source* interfaces.

```
coclass TangramModel
{
    [default] interface ITangramModel ;
    interface ITangramTransform ;
    interface IConnectionPointContainer ;

    // Outgoing source interface.
    [source] interface ITangramModelEvent ;
};
```

The COM source interface declaration is similar to the **uses** statement in this submission; non-source interfaces are similar to **provides** statements. While the **coclass** must declare a default interface, this submission allows the component to support operations of its own.

D.5 Rapide

The **provides** and **uses** statements in this submission are similar to the Interface Connection Architecture implemented in Rapide [Rapide 97] and discussed in [Luckham 95]. The Rapide Interface Connection Architecture applies *provides* and *requires* statements to individual functions in a class declaration. Class instances are connected via a connect statement in which a *requires* method of one object is connected to a *provided* method of another object.

For example:

```
class Parser is
provides:
    function Initialize();
    function FileName() return String;
requires
    function Semantize(Tree);
    function Generate(Tree);
specification ...
end Parser;

class Semanticizer is
provides:
    function Semantize(Tree);
    function Incremental_Semantize(Context : Tree; Addition : Tree);
requires:
    function FileName() return String;
specification ...
end Semanticizer;

P: Parser; S: Semanticizer;

Connect
    P.Semantize to S.Semantize;
    S.FileName to P.FileName;
```

This submission specifies a similar notion of explicit dependency specification. The difference being that we specify dependencies with respect to interfaces rather that individual methods.

References E

[Agesen 97] Ole Agesen, Stephen N. Freund, John C. Mitchell, “Adding Type Parameterization to the Java Language”, Proceedings of the 1997 OOPSLA--Conference on Object-Oriented Programming Systems, Languages and Applications.

[Englander 97] Robert Englander, Developing Java Beans, O’Reilly & Associates, Sebastopol, CA, 1997.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[Garg 98] Rohit Garg, Enterprise JavaBeans to CORBA Mapping 1.0, Sun Microsystems, http://java.sun.com/products/ejb/ejb-corba.10.pdf, 1998

[Hamilton 97] Graham Hamilton (Editor), JavaBeans Specification 1.01, Sun Microsystems, http://www.javasoft.com/beans/docs/beans.101.pdf, 1997.

[Liskov 95] Mark Kay, Robert Gruber, Barbara Liskov, “Subtypes vs. Where Clauses: Constraining Parametric Polymorphism”, Proceedings of the 1995 OOPSLA--Conference on Object-Oriented Programming Systems, Languages and Applications.

[Luckham 95] David C. Luckham, James Vera, Sigurd Meldal, “Three Concepts of System Architecture”, Unpublished Manuscript, Stanford University CS Technical Report, CSL-TR-95-674, July 19, 1995.

[Matena 98] Vlada Matena, Mark Hapner, Enterprise JavaBeans Specification 1.0, Sun Microsystems, http://java.sun.com/products/ejb/ejb.10.pdf, 1998

[Myers 97] Andrew C. Myers, Joseph A. Bank, Barbara Liskov, “Parameterized Types for Java”, Proceedings of the 1997 ACM Symposium on Principles of Programming Languages (POPL).

[Rapide 97] The Stanford Rapide Project, http://poset.stanford.edu/rapide/rapide-pubs.html.

[Rogerson 97] Dale Rogerson, Inside COM, Microsoft Press, Redmond WA, 1997.