CORBA Components

Components 1.1 RTF

September 2002
Version 3.0
ptc/02-08-03

An Adopted Specification of the Object Management Group, Inc.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo¨, CORBA¨, CORBA Academy¨, The Information Brokerage¨, XMI¨ and IIOP¨ are registered trademarks of the Object Management Group. OMGª, Object Management Groupª, CORBA logosª, OMG Interface Definition Language (IDL)ª, The Architecture of Choice for a Changing Worldª, CORBAservicesª, CORBAfacilitiesª, CORBAmedª, CORBAnetª, Integrate 2002ª, Middleware That's Everywhereª, UMLª, Unified Modeling Languageª, The UML Cube logoª, MOFª, CWMª, The CWM Logoª, Model Driven Architectureª, Model Driven Architecture Logosª, MDAª, OMG Model Driven Architectureª, OMG MDAª and the XMI Logoª are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

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

Contents

1. Component Model 1-1

- 1.1 Component Model 1-2
  - 1.1.1 Component Levels 1-2
  - 1.1.2 Ports 1-2
  - 1.1.3 Components and Facets 1-3
  - 1.1.4 Component Identity 1-4
  - 1.1.5 Component Homes 1-5
- 1.2 Component Definition 1-5
- 1.3 Component Declaration 1-6
  - 1.3.1 Basic Components 1-6
  - 1.3.2 Equivalent IDL 1-6
  - 1.3.3 Component Body 1-7
- 1.4 Facets and Navigation 1-8
  - 1.4.1 Equivalent IDL 1-8
  - 1.4.2 Semantics of Facet References 1-8
  - 1.4.3 Navigation 1-9
  - 1.4.4 Provided References and Component Identity . 1-12
  - 1.4.5 Supported interfaces 1-13
- 1.5 Receptacles 1-15
  - 1.5.1 Equivalent IDL 1-15
  - 1.5.2 Behavior 1-16
  - 1.5.3 Receptacles Interface 1-18
- 1.6 Events 1-21
  - 1.6.1 Event types 1-21
  - 1.6.2 EventConsumer Interface 1-22

1.6.3 Event Service Provided by Container 1-23
1.6.4 Event SourcesÑPublishers and Emitters 1-23
1.6.5 Publisher 1-24
1.6.6 Emitters 1-26
1.6.7 Event Sinks 1-27
1.6.8 Events interface 1-29
1.7 Homes 1-32
1.7.1 Equivalent Interfaces 1-33
1.7.2 Primary Key Declarations 1-36
1.7.3 Explicit Operations in Home Definitions 1-37
1.7.4 Home inheritance 1-38
1.7.5 Semantics of Home Operations 1-39
1.7.6 CCMHome Interface 1-41
1.7.7 KeylessCCMHome Interface 1-42
1.8 Home Finders 1-42
1.9 Component Configuration 1-44
1.9.1 Exclusive Configuration and Operational Life Cycle Phases 1-46
1.10 Configuration with Attributes 1-47
1.10.1 Attribute Configurators 1-47
1.10.2 Factory-based Configuration 1-48
1.11 Component Inheritance 1-51
1.11.1 CCMObject Interface 1-52
1.12 Conformance Requirements 1-54
1.12.1 A Note on Tools 1-55
1.12.2 Changes to Object Services 1-55

2. OMG CIDL Syntax and Semantics 2-1
2.1 Overview 2-2
2.2 Lexical Conventions 2-3
2.2.1 Keywords 2-3
2.3 OMG CIDL Grammar 2-3
2.4 OMG CIDL Specification 2-5
2.5 Composition Definition 2-5
2.5.1 Life Cycle Category and Constraints 2-6
2.6 Catalog Usage Declaration 2-7
2.7 Home Executor Definition 2-8
2.8 Home Implementation Declaration 2-9
2.9 Storage Home Binding 2-9

2.10 Home Persistence Declaration 2-10
2.11 Executor Definition 2-10
2.12 Segment Definition 2-10
2.13 Segment Persistence Declaration 2-11
2.14 Facet Declaration 2-12
2.15 Feature Delegation Specification 2-12
2.16 Abstract Storage Home Delegation Specification 2-13
2.17 Executor Delegation Specification 2-15
2.18 Abstract Spec Declaration. 2-16
2.19 Proxy Home Declaration 2-16

3. CCM Implementation Framework 3-1

3.1 Introduction 3-1
3.2 Component Implementation Framework (CIF) Architecture 3-1
  3.2.1 Component Implementation Definition Language (CIDL) 3-2
  3.2.2 Component persistence and behavior 3-2
  3.2.3 Implementing a CORBA Component 3-2
  3.2.4 Behavioral elements: Executors 3-2
  3.2.5 Unit of implementation : Composition 3-3
  3.2.6 Composition structure 3-4
  3.2.7 Compositions with Managed Storage 3-11
  3.2.8 Relationship between Home Executor and Abstract Storage Home 3-13
  3.2.9 Executor Definition 3-26
  3.2.10 Proxy Homes 3-35
  3.2.11 Component Object References 3-36
3.3 Language Mapping 3-38
  3.3.1 Overview 3-38
  3.3.2 Common Interfaces 3-39
  3.3.3 Mapping Rules 3-40

4. The Container Programming Model 4-1

4.1 Introduction 4-2
  4.1.1 External API Types 4-4
  4.1.2 Container API Type 4-4
  4.1.3 CORBA Usage Model 4-4
  4.1.4 Component Categories 4-5
4.2 The Server Programming Environment 4-5
  4.2.1 Component Containers 4-5

4.2.2 CORBA Usage Model 4-6
4.2.3 Component Factories 4-8
4.2.4 Component Activation 4-8
4.2.5 Servant Lifetime Management 4-8
4.2.6 Transactions 4-9
4.2.7 Security 4-11
4.2.8 Events 4-12
4.2.9 Persistence 4-13
4.2.10 Application Operation Invocation 4-15
4.2.11 Component Implementations 4-15
4.2.12 Component Levels 4-15
4.2.13 Component Categories 4-16

4.3 Server Programming Interfaces - Basic Components 4-21
4.3.1 Component Interfaces 4-21
4.3.2 Interfaces Common to both Container API Types 4-22
4.3.3 Interfaces Supported by the Session Container API Type 4-27
4.3.4 Interfaces Supported by the Entity Container API Type 4-30

4.4 Server Programming Interfaces - Extended Components . 4-33
4.4.1 Interfaces Common to both Container API Types 4-33
4.4.2 Interfaces Supported by the Session Container API Type 4-36
4.4.3 Interfaces Supported by the Entity Container API Type 4-38

4.5 The Client Programming Model 4-45
4.5.1 Component-aware Clients 4-46
4.5.2 Component-unaware Clients 4-50

5. Integrating with Enterprise JavaBeans 5-1

5.1 Introduction 5-1

5.2 Enterprise JavaBeans Compatibility Objectives and Requirements 5-3

5.3 CORBA Component Views for EJBs 5-4
5.3.1 Mapping of EJB to Component IDL definitions 5-5
5.3.2 Translation of CORBA Component requests into EJB requests 5-9
5.3.3 Interoperability of the View 5-10
5.3.4 CORBA Component view Example 5-12

5.4 EJB views for CORBA Components 5-14
  5.4.1 Mapping of Component IDL to Enterprise JavaBeans specifications 5-14
  5.4.2 Translation of EJB requests into CORBA Component Requests 5-17
  5.4.3 Interoperability of the View 5-19
  5.4.4 Example 5-21
5.5 Compliance with the Interoperability of Integration Views 5-22
5.6 Comparing CCM and EJB 5-23
  5.6.1 The Home Interfaces 5-23
  5.6.2 The Component Interfaces 5-25
  5.6.3 The Callback Interfaces 5-26
  5.6.4 The Context Interfaces 5-28
  5.6.5 The Transaction Interfaces 5-29
  5.6.6 The Metadata Interfaces 5-30
6. Packaging and Deployment 6-1
6.1 Introduction 6-1
6.2 Component Packaging 6-2
6.3 Software Package Descriptor 6-2
  6.3.1 A softpkg Descriptor Example 6-3
  6.3.2 The Software Package Descriptor XML Elements 6-4
6.4 CORBA Component Descriptor 6-16
  6.4.1 Component Feature Description 6-16
  6.4.2 Deployment Information 6-17
  6.4.3 CIDL Compiler Responsibilities 6-18
  6.4.4 CORBA Component Descriptor Example 6-18
  6.4.5 The CORBA Component Descriptor XML Elements 6-20
6.5 Component Assembly Packaging 6-39
6.6 Component Assembly File 6-40
6.7 Component Assembly Descriptor 6-40
  6.7.1 Component Assembly Descriptor Example . . . 6-40
  6.7.2 Component Assembly Descriptor XML Elements 6-43
6.8 Property File Descriptor 6-61
  6.8.1 Property File Example 6-61
  6.8.2 Property File XML Elements 6-62
6.9 Component Deployment 6-66

6.9.1 Participants in Deployment 6-67
6.9.2 ComponentInstallation Interface 6-71
6.9.3 AssemblyFactory Interface 6-72
6.9.4 Assembly Interface 6-73
6.9.5 ServerActivator interface 6-74
6.9.6 ComponentServer interface 6-75
6.9.7 Container interface 6-76
6.9.8 Component Entry Points (Component Home Factories) 6-78

7. XML DTDs 7-1

7.1 softpkg.dtd 7-1
7.2 corbacomponent.dtd 7-5
7.3 properties.dtd 7-11
7.4 componentassembly.dtd 7-13

8. Interface Repository Metamodel 8-1

8.1 Introduction 8-1
8.1.1 BaseIDL Package 8-2
8.1.2 ComponentIDL Package 8-14
8.2 Conformance Criteria 8-26
8.2.1 Conformance Points 8-27
8.3 MOF DTDs and IDL for the Interface Repository Metamodel 8-27
8.3.1 XMI DTD 8-27
8.3.2 IDL for the BaseIDL Package 8-59
8.3.3 IDL for the ComponentIDL Package 8-80

9. CIF Metamodel 9-1

9.1 CIF Package 9-1
9.2 Classes and Associations 9-2
9.3 Conformance Criteria 9-5
9.3.1 Conformance Points 9-5
9.4 MOF DTDs and IDL for the CIF Metamodel 9-6
9.4.1 XMI DTD 9-6
9.4.2 IDL for the CIF Package 9-7

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

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

OMGÕs objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industryÕs premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/.

Intended Audience

The architecture and specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. As defined by the Object Management Group (OMG) in the *Object Management Architecture Guide,* the ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.

- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification.*
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects,** which are products of a single vendor on in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMGÕs technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the Object Services.
- *CORBAfacilities: Common Facilities Architecture* contains the architecture for Common Facilities.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

You can download the OMG formal documents free-of-charge from our web site in PostScript and PDF format. Please note the OMG address and telephone numbers below:

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

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users arenÕt required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in the *C++ Language Mapping Specification*.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to CORBA Core, the Interworking Architecture chapter, ÒCompliance to COM/CORBA Interworking.Ó

Typographical Conventions

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

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

`Courier bold` - Programming language elements.

Helvetica - Exceptions

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

Acknowledgements

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

¥ Computational Physics, Inc.
¥ Laboratoire dÕInformatique Fondamentale de Lille

Component Model

1

Note Ð This specification is based on the Adopted CORBA Components Specification (formal/2002-06-65). Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter describes the semantics of the CORBA Component Model (CCM) and the conformance requirements for vendors.

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒComponent ModelÓ	1-2
ÒComponent DefinitionÓ	1-5
ÒComponent DeclarationÓ	1-6
ÒFacets and NavigationÓ	1-8
ÒReceptaclesÓ	1-15
ÒEventsÓ	1-21
ÒHomesÓ	1-33
ÒHome FindersÓ	1-43
ÒComponent ConfigurationÓ	1-45

Section Title	Page
ÒConfiguration with AttributesÓ	1-47
ÒComponent InheritanceÓ	1-51
ÒConformance RequirementsÓ	1-54

1.1 Component Model

Component is a basic meta-type in CORBA. The component meta-type is an extension and specialization of the object meta-type. Component types are specified in IDL and represented in the Interface Repository. A component is denoted by a component reference, which is represented by 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, they are designed to be 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. 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. 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.

1.1.1 Component Levels

There are two levels of components: *basic* and *extended*. Both are managed by component homes, but they differ in the capabilities they can offer. Basic components essentially provide a simple mechanism to ÒcomponentizeÓ a regular CORBA object. Extended components, on the other hand, provide a richer set of functionality.

> *A basic component is very similar in functionality to an EJB as defined in the Enterprise JavaBeans 1.1 specification. This allows much easier mapping and integration at this level.*

1.1.2 Ports

Components support a variety of surface features through which clients and other elements of an application environment may interact with a component. 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 componentÕs 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. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

Basic components are not allowed to offer facets, receptacles, event sources and sinks. They may only offer attributes.

Extended components may offer any type of port.

1.1.3 Components and Facets

A component can provide multiple object references, called *facets*, which are capable of supporting distinct (i.e., unrelated by inheritance) IDL interfaces. The component has a single distinguished reference whose interface conforms to the component definition. This reference supports an interface, called the componentÕs *equivalent interface*, that manifests the componentÕs surface features to clients. The equivalent interface allows clients to navigate among the componentÕs facets, and to connect to the componentÕs ports.

Basic components cannot support facets, therefore attempts to navigate to other facets will always fail. The equivalent interface of a basic component is the only object available with which a client may interact.

The other interfaces provided by the component are referred to as *facets*. Figure 1-1 illustrates the relationship between the component and its facets.

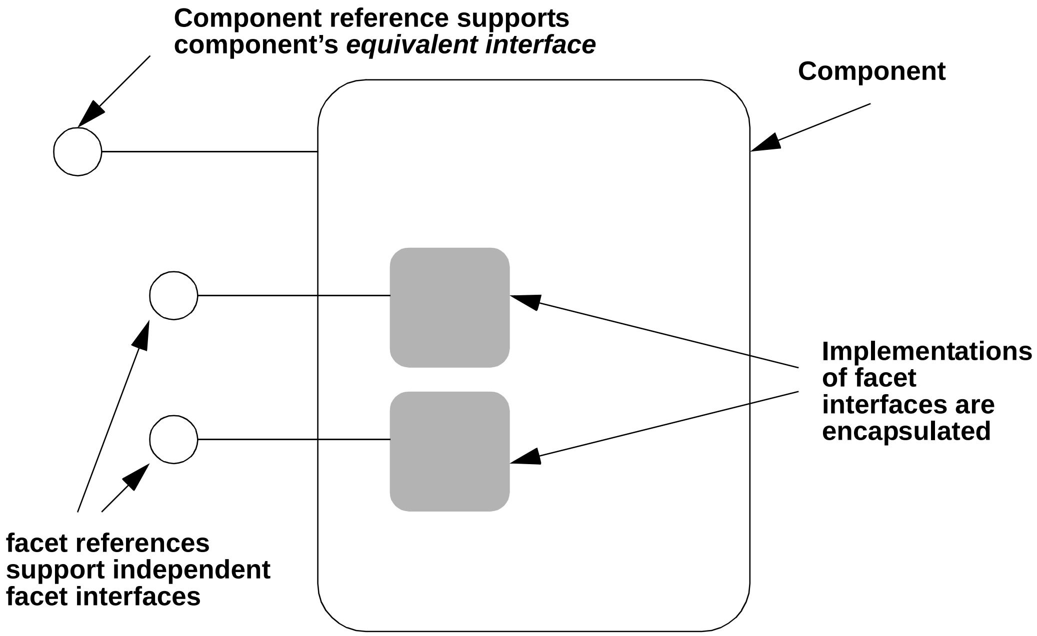


Figure 1-1 Component Interfaces and Facets

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 ÒpartsÓ 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 is bounded by the life cycle of its owning component.

1.1.4 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 componentÕs references. The definition of ÒsameÓ component instance is ultimately up to the component implementor, in that they may provide a customized implementation of this operation. However, a component framework shall provide standard implementations that constitute *de facto* definitions of ÒsamenessÓ 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 componentÕs 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.

1.1.5 Component Homes

A *component home* is 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). A home must be declared for every component declaration.

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.

1.2 Component Definition

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 IDL grammar for components may be found in CORBA Core, *OMG IDL Syntax and Semantics* chapter.

1.3 Component Declaration

1.3.1 Basic Components

Basic components cannot avail themselves of certain features in the model. In particular, they cannot inherit from other components, nor can they provide or use interfaces, or make any event declarations. A basic component is declared using a restricted version of a **<component_dcl>**. See CORBA Core, *OMG IDL Syntax and Semantics* chapter, Section 3.17.1, ÒComponentÓ for the syntax.

To avoid ambiguity between basic and extended definitions, any component declaration that matches the following pattern is a basic component:

```
“component” <identifier> [<supported_interface_spec>]
            “{“ {<attr_dcl> “;”}* “}”
```

Ideally the syntax should explicitly represent these rules. However this can only be achieved by introducing a new keyword to distinguish between basic and extended components. It was felt that an extra keyword would cause problems in the future, as the distinction between basic and extended components gets blurred. This blurring may occur due to future development of both the CORBA Component Model and the Enterprise JavaBeans specifications.

1.3.2 Equivalent IDL

The client mappings; that is, mappings of the externally-visible component features for component declarations are described in terms of *equivalent IDL*.

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.

Since basic components are essentially a profile, no specific rules are defined for them.

1.3.2.1 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::CCMObject { É };
```

1.3.2.2 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::CCMObject,
<interface_name_1>, <interface_name_2> { É };
```

Supported interfaces are described in detail in Section 1.4.5, ÒSupported interfaces,Ó on page 1-13.

1.3.2.3 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> { É }
```

1.3.2.4 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> { É };
```

1.3.3 Component Body

A component forms a naming scope, nested within the scope in which the component is declared.

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

1.4 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. A component may exhibit zero or more facets.

1.4.1 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 operation defined on the equivalent interface:

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

The mechanisms for navigating among a componentÕs facets are described in Section 1.4.3, ÒNavigation,Ó on page 1-9. The relationships between the component identity and the facet references, and assumptions regarding facet references, are described in Section 1.4.4, ÒProvided References and Component Identity,Ó on page 1-12. 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 the *OMG CIDL Syntax and Semantics* chapter.

1.4.2 Semantics of Facet References

Clients of a component instance can obtain a reference to a facet by invoking the **provide_<*name*>** operation on the equivalent 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 shall be prepared to return object references for facets throughout the instanceÕs 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 shall 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 shall be **TRUE**, and legal operations of the facet interface shall 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 Section 1.4.4, ÒProvided References and Component Identity,Ó on page 1-12 and Section 1.4.3, ÒNavigation,Ó on page 1-9.

1.4.3 Navigation

Navigation among a componentÕs 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 equivalent 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::CCMObject**). Other operations on the **Navigation** interface (i.e., **get_all_facets** and **get_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**, **get_all_facets**, and **get_named_facets**.
- For components, such as basic components, that do not provide interfaces, only the generic navigation operations are available on the equivalent interface. The behavior of these operations, where there are no facets to navigate to, is defined below.

The detailed descriptions of these mechanisms follow.

1.4.3.1 get_component()

```
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**) the operation name in GIOP request corresponding to **get_component** shall be Ò**_component**Ó. An implementation of **get_component** is a required element of the CORBA core, even if the ORB does not provide an implementation of CORBA components. Thus component vendors that are not also ORB vendors can rely on the availability of this capability in a compliant ORB.

1.4.3.2 Component-specific provide operations

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

1.4.3.3 Navigation interface on the component

As described in Section 1.3, ÒComponent Declaration,Ó on page 1-6 all component interfaces implicitly inherit directly or indirectly from **CCMObject**, which inherits from **Components::Navigation**. The definition of the **Components::Navigation** interface is as follows:

```
module Components {

    typedef string FeatureName;

    typedef sequence<FeatureName> NameList;

    valuetype PortDescription
    {
        public FeatureName name;
        public CORBA::RepositoryId type_id;
    };

    valuetype FacetDescription : PortDescription
    {
        public Object facet_ref;
    };
    typedef sequence<FacetDescription> FacetDescriptions;

    exception InvalidName { };

    interface Navigation {

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

        FacetDescriptions get_all_facets();
```

```
        FacetDescriptions get_named_facets (in NameList names)
                    raises (InvalidName);

        boolean same_component (in Object 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; that is, 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 componentÕs facets, the InvalidName exception shall be raised. A component that does not provide any facets (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the InvalidName exception.

get_all_facets

The **get_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 shall contain descriptions and references for all of the facets in the componentÕs inheritance hierarchy. The order in which these values occur in the sequence is not specified. A component that does not provide any facets (e.g., a basic component) shall return a sequence of length zero.

get_named_facets

The **get_named_facets** operation returns a sequence of described references (identical to the sequence returned by **get_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. A component that does not provide any facets (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the InvalidName exception.

The **same_component** operation on **Navigation** is described in Section 1.4.4, ÒProvided References and Component Identity,Ó on page 1-12.

1.4.3.4 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**, **get_all_facets**, and **get_named_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 shall 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:

Note Ð Issue 5506

```
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(ÒbÓ);
```

1.4.4 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 Òsame component instanceÓ means. The skeletons generated by the Component Implementation Framework shall provide an implementation of **same_component**, where Òsame instanceÓ 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 shall 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 shall provide an implementation of **same_component** for facets that inherit the **Navigation** interface.

1.4.5 Supported interfaces

A component definition may optionally support one or more interfaces, or in the case of extended components, 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 **CCMObject** and any supported interfaces, as follows:

interface <*component_name*>
: Components::CCMObject, <*interface_name*> { É };

The component implementation shall supply implementations of operations defined on supported interfaces. Clients shall be able to widen a reference of the componentÕs equivalent interface type to the type of any of the supported interfaces. Clients shall also be able to narrow a reference of type **CCMObject** to the type of any of the componentÕs 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 shall be derived from KeylessCCMHome, supporting the create_component operation, which returns a reference of type CCMObject. This reference shall be able to be narrowed directly from CCMObject to I:

```
// java
...
M.AManager aHome = ...; // get A's home
org.omg.Components.CCMObject 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.CCMObject, 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 interface. 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 AÕs home, but doesnÕt statically know about AÕs interface or home interface:

```
org.omg.Components.KeylessCCMHome genericHome =
... // get A’s home;
org.omg.Components.CCMObject 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.

1.5 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. A component may exhibit zero or more receptacles.

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.

1.5.1 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>Connec-
tions;

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> ( );
```

1.5.2 Behavior

1.5.2.1 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 1 - *OMG CIDL Syntax and Semantics*. 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 shall hold a copy of the reference until it is explicitly disconnected.

Simplex receptacles

If a receptacleÕs **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 shall raise the AlreadyConnected exception.

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

Multiplex receptacles

If a receptacleÕs **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 shall raise the ExceededConnectionLimit exception.

The component implementation may refuse to accept the connection for arbitrary reasons. If it does so, the connection operation shall 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 1.5.2.4, ÒCookie type,Ó on page 1-18.Ó

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

1.5.2.2 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 shall raise a NoConnection 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 shall raise an InvalidConnection exception.

1.5.2.3 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.

1.5.2.4 Cookie type

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

```
module Components {
    valuetype Cookie {
        private CORBA::OctetSeq 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 shall be truncatable to **Cookie**, and the information preserved in the **cookieValue** octet sequence shall be sufficient for the receptacle implementation to identify the cookie and its associated connected reference.

1.5.3 Receptacles Interface

The **Receptacles** interface provides generic operations for connecting to a componentÕs receptacles. The **CCMObject** interface is derived from **Receptacles**. For components, such as basic components, that do not use interfaces, only the generic receptacles operations are available on the equivalent interface. The default behavior in such cases is defined below.

The **Receptacles** interfaces is defined by the following IDL:

```
module Components {

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

```
typedef sequence<ConnectionDescription> ConnectionDescriptions;

valuetype ReceptacleDescription : PortDescription
{
    public boolean is_multiple;
    public ConnectionDescriptions connections;
};
typedef sequence<ReceptacleDescription> ReceptacleDescriptions;

exception ExceededConnectionLimit { };

exception CookieRequired { };

interface Receptacles {

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

Note Ð Issue 4983

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

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

        ReceptacleDescriptions get_all_receptacles ();

        ReceptacleDescriptions get_named_receptacles (
            in NameList names) 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 receptacleÕs **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.
- A component that does not have any receptacles (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the InvalidName exception.

disconnect

Note Ð Issue 4983

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. In both cases, the **disconnect** operation returns the previously connected object reference. 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.
- A component that does not have any receptacles (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the InvalidName exception.

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. A component that does not have any receptacles (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the InvalidName exception.

get_all_receptacles

The **get_all_receptacles** operation returns information about all receptacle ports in the component's inheritance hierarchy as a sequence of **ReceptacleDescription** values. The order in which these values occur in the sequence is not specified. For components that do not have any receptacles (e.g., a basic component), this operation returns a sequence of length zero.

get_named_receptacles

The **get_named_receptacles** operation returns information about all receptacle ports denoted by the **names** parameter as a sequence of **ReceptacleDescription** values. The order in which these values occur in the sequence is not specified. If any name in the **names** parameter is not a valid name for a receptacle in the component's inheritance hierarchy, the operation raises the InvalidName exception. A component that does not provide any receptacles (e.g., a basic component) will have no valid name parameter to this operation and thus shall always raise the InvalidName exception.

1.6 Events

The CORBA component model supports a publish/subscribe event model. The event model for CORBA components is designed to be compatible with CORBA notification, as defined in OMG document telecom/98-11-01. The interfaces exposed by the component event model provide a simple programming interface whose semantics can be mapped onto a subset of CORBA notification semantics.

1.6.1 Event types

IDL contains event type declarations, which are a restricted form of value type declarations. They are for the use in the CORBA Component event model.

Since the underlying implementation of the component event mechanism provided by the container is CORBA notification, event values shall be inserted into instances of the **any** type. The resulting **any** values shall be inserted into a CORBA notification structured event. The mapping between a component event and a notification event is implemented by the container.

1.6.1.1 Equivalent IDL

For the declaration of event types of the following form:

```
module <module_name> {

    valuetype A { <A_state_members> };

    eventtype B : A { <B_state_members> };

    eventtype C : B { <C_state_members> };
};
```

The following equivalent IDL is implied:

```
module <module_name> {

    valuetype A { <A_state_members> };

    valuetype B : A, ::Components::EventBase {
        <B_state_members>
    };

    interface BConsumer : ::Components::EventConsumerBase {
        void push_B (in B the_b);
    };

    valuetype C : B {
        <C_state_members>
    };

    interface CConsumer : BConsumer {
        void push_C (in C the_c);
    };
};
```

As shown above the first event type in the inheritance chain introduces the inheritance from **Components::EventBase** into the inheritance chain for the equivalent value types. The same rule applies for the equivalent consumer interfaces and **Components::EventConsumerBase**. Consumer interfaces are in the same inheritance relation as the event types, where they origin.

1.6.1.2 EventBase

The module **Components** contains the following abstract value type definition:

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

It serves as base type for value types derived via the Equivalent IDL mapping for event types.

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** shall 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.

1.6.2 *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 shall be raised. The **expected_event_type** member of the exception contains the **RepositoryId** 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).

1.6.3 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.

1.6.4 Event SourcesÑPublishers and Emitters

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

There are two categories of event sources, *emitters* and *publishers*. Both are implemented using event channels supplied by the container. An emitter can be connected to at most one proxy provider by the container. A publisher can be connected through the channel to an arbitrary number of consumers, who are said to *subscribe* to the publisher event source. A component may exhibit zero or more emitters and publishers.

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 are delegated to an event channel supplied by the container at run time. Other event sources, however, may use the same channel. Events pushed from an emitter are then pushed by the container into the 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.*

1.6.5 Publisher

1.6.5.1 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:

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 61-241).

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

~~interface <component_name> : Components::CCMObject {~~

    ~~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);~~
        ~~};~~
~~};~~
~~};~~
```

```
module <module_name> {
    interface <component_name> : Components::CCMObject {
        Components::Cookie subscribe_<source_name> (
                        in <event_type>Consumer consumer)
                        raises (Components::ExceededConnectionLimit);

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

};

1.6.5.2 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 shall 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 shall raise an `InvalidConnection` exception.

1.6.6 Emitters

1.6.6.1 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:

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 61-243).

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

~~**interface <*component_name*> : Components::CCMObject {**~~

~~**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);~~
            ~~};~~
    ~~};~~
~~};~~

module <module_name> {
    interface <component_name> : Components::CCMObject {
        void connect_<source_name> (
                                    in <event_type>Consumer consumer)
                                    raises (Components::AlreadyConnected);

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

1.6.6.2 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 shall raise 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.

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 ÒEventConsumersÓ to the component type name) is defined in the same scope as the componentÕs 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 ÒConsumerÓ 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.

1.6.7 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. A component may exhibit zero or more consumers.

> *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.*

1.6.7.1 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:

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 61-245).

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

    ~~interface <component_name> : Components::CCMObject {~~
        ~~<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);~~
            ~~};~~
        ~~};~~
~~};~~
```

```
module <module_name> {
    interface <component_name> : Components::CCMObject {
        <event_type>Consumer get_consumer_<sink_name>();
    };
};
```

1.6.7.2 *Event sink operations*

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

1.6.8 *Events interface*

The **Events** interface provides generic access to event sources and sinks on a component. **CCMObject** is derived from **Events**. For components, such as basic components, that do not declare participation in events, only the generic **Events** operations are available on the equivalent interface. The default behavior in such cases is described below.

The **Events** interface is described as follows:

```
module Components {

    exception InvalidName { };
    exception InvalidConnection { };
    exception AlreadyConnected { };
    exception NoConnection { };

    valuetype ConsumerDescription : PortDescription
    {
        public EventConsumerBase consumer;
    };
    typedef sequence<ConsumerDescription> ConsumerDescriptions;

    valuetype EmitterDescription : PortDescription
    {
        public EventConsumerBase consumer;
    };
    typedef sequence<EmitterDescription> EmitterDescriptions;

    valuetype SubscriberDescription
    {
```

```
        public Cookie ck;
        public EventConsumerBase consumer;
    };
    typedef sequence<SubscriberDescription> SubscriberDescriptions;

    valuetype PublisherDescription : PortDescription
    {
        public SubscriberDescriptions consumers;
    };
    typedef sequence<PublisherDescription> PublisherDescriptions;

    interface Events {
        EventConsumerBase get_consumer (in FeatureName sink_name)
                                raises (InvalidName);
        Cookie subscribe (in FeatureName publisher_name,
                                in EventConsumerBase subscriber)
                                raises (InvalidName, InvalidConnection,
                                        ExceededConnectionLimit);
```

Note Ð Issue 4983

```
        ~~void~~
        EventConsumer unsubscribe (in FeatureName publisher_name,
                                in Cookie ck)
                                raises (InvalidName, InvalidConnection);
        void connect_consumer (in FeatureName emitter_name,
                                in EventConsumerBase consumer)
                                raises (InvalidName, AlreadyConnected,
                                        InvalidConnection);
        EventConsumerBase disconnect_consumer (
                                in FeatureName source_name)
                                raises (InvalidName, NoConnection);
        ConsumerDescriptions get_all_consumers ();
        ConsumerDescriptions get_named_consumers (
                                in NameList names)
                                raises (InvalidName);
        EmitterDescriptions get_all_emitters ();
        EmitterDescriptions get_named_emitters (in NameList names)
                                raises (InvalidName);
        PublisherDescriptions get_all_publishers ();
        PublisherDescriptions get_named_publishers (in NameList names)
                                raises (InvalidName);
    };
};
```

get_consumer

The **get_consumer** operation returns the **EventConsumerBase** interface for the sink specified by the **sink_name** parameter. If the **sink_name** parameter does not specify a valid event sink on the component, the operation raises the InvalidName exception. A component that does not have any sinks (e.g., a basic component) will have no valid **sink_name** parameter to this operation and thus shall always raise the InvalidName exception.

subscribe

The **subscribe** operation associates the subscriber denoted by the **subscriber** parameter with the event source specified by the **publisher_name** parameter. If the **publisher_name** 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. A component that does not have any event sources (e.g., a basic component) will have no valid **publisher_name** parameter to this operation and thus shall always raise the InvalidName exception. If the object reference in the **subscriber** parameter does not support the consumer interface of the eventtype declared in the **publishes** statement, the InvalidConnection exception is raised. If the implementation-defined limit to the number of subscribers is exceeded, the ExceededConnectionLimit exception is raised.

unsubscribe

Note Ð Issue 4983

The **unsubscribe** operation disassociates the subscriber associated with **ck** parameter with the event source specified by the **publisher_name** parameter, and returns the reference to the subscriber. If the **publisher_name** 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. A component that does not have any event sources (e.g., a basic component) will have no valid **publisher_name** parameter to this operation and thus shall always raise the InvalidName exception.

connect_consumer

The **connect_consumer** operation associates the consumer denoted by the **consumer** parameter with the event source specified by the **emitter_name** parameter. If the **emitter_name** 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. If the object reference in the **consumer** parameter does not support the consumer interface of the eventtype declared in the **emits** statement, the InvalidConnection exception is raised. The cookie return value can be used to disconnect from the source. A component that does not have any event sources (e.g., a basic component) will have no valid **emitter_name** parameter to this operation and thus shall always raise the InvalidName exception.

disconnect_consumer

The **disconnect_consumer** operation disassociates the currently connected consumer from the event source specified by the **emitter_name** parameter, returning a reference to the disconnected consumer. If the **emitter_name** 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. A component that does not have any event sources (e.g., a basic component) will have no valid **emitter_name** parameter to this operation and thus shall always raise the InvalidName exception.

get_all_consumers

The **get_all_consumers** operation returns information about all consumer ports in the component's inheritance hierarchy as a sequence of **ConsumerDescription** values. The order in which these values occur in the sequence is not specified. For components that do not consume any events (e.g., a basic component), this operation returns a sequence of length zero.

get_named_consumers

The **get_named_consumers** operation returns information about all consumer ports denoted by the **names** parameter as a sequence of **ConsumerDescription** values. The order in which these values occur in the sequence is not specified. If any name in the **names** parameter is not a valid name for an event sink in the component's inheritance hierarchy, the operation raises the InvalidName exception. A component that does not provide any consumers (e.g., a basic component) will have no valid name parameter to this operation and thus shall always raise the InvalidName exception.

get_all_emitters

The **get_all_emitters** operation returns information about all emitter ports in the component's inheritance hierarchy as a sequence of **EmitterDescription** values. The order in which these values occur in the sequence is not specified. For components that do not emit any events (e.g., a basic component), this operation returns a sequence of length zero.

get_named_emitters

The **get_named_emitters** operation returns information about all emitter ports denoted by the **names** parameter as a sequence of **EmitterDescription** values. The order in which these values occur in the sequence is not specified. If any name in the **names** parameter is not a valid name for an emitter port in the component's inheritance hierarchy, the operation raises the InvalidName exception. A component that does not provide any emitters (e.g., a basic component) will have no valid name parameter to this operation and thus shall always raise the InvalidName exception.

get_all_publishers

The **get_all_publishers** operation returns information about all publisher ports in the component's inheritance hierarchy as a sequence of **PublisherDescription** values. The order in which these values occur in the sequence is not specified. For components that do not publish any events (e.g., a basic component), this operation returns a sequence of length zero.

get_named_publishers

The **get_named_publishers** operation returns information about all publisher ports denoted by the **names** parameter as a sequence of **PublisherDescription** values. The order in which these values occur in the sequence is not specified. If any name in the **names** parameter is not a valid name for a publisher port in the component's inheritance hierarchy, the operation raises the InvalidName exception. A component that does not provide any publishers (e.g., a basic component) will have no valid name parameter to this operation and thus shall always raise the InvalidName exception.

1.7 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.
- The presence of a primary key specification in a home definition causes homeÕs 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 1.7.1.2, ÒHome definitions with primary keys,Ó on page 1-34.

1.7.1 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 operationsÑcreation, 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 goalsÑuniformity and type safetyÑare 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.*

1.7.1.1 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::CCMHome {
    <equivalent_explicit_operations>
};

interface <home_name>Implicit : Components::KeylessCCMHome {
    <component_type> create() raises(CreateFailure);
};

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 1.7.3, ÒExplicit Operations in Home Definitions,Ó on page 1-37.Ó

create

This operation creates a new component instance of the type managed by the home. The CreateFailure exception is raised if any application errors are encountered in home creation.

1.7.1.2 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 : Components::CCMHome {
    <equivalent_explicit_operations>
};

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

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

    void remove (in <key_type> key)
    raises (Components::RemoveFailure, 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 1.7.3, ÒExplicit Operations in Home Definitions,Ó on page 1-37.

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 shall raise the InvalidKey exception. All other error conditions may raise the CreateFailure exception.

find_by_primary_key

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 shall raise the InvalidKey exception. All other error conditions may raise the FinderFailure exception.

remove

This operation removes the component identified by the specified key value. Subsequent requests to any of the componentÕs facets shall raise an 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 shall raise the InvalidKey exception. All other error conditions may raise the RemoveFailure exception.

1.7.1.3 Supported interfaces

A home definition may optionally support one or more interfaces. When a home definition header includes a **supports** clause as follows:

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

The resulting explicit interface inherits both **CCMHome** and any supported interfaces, as follows:

```
interface <home_name>Explicit : Components::CCMHome,
                                     <interface_name> {
    <equivalent_explicit_operations>
};
```

The home implementation shall supply implementations of operations defined on supported interfaces. Clients shall be able to widen a reference of the homeÝs resulting explicit or equivalent interface type to the type of any of the supported interfaces. Clients shall also be able to narrow a reference of type **CCMHome** to the type of any of the homeÝs supported interfaces.

1.7.2 Primary Key Declarations

Primary key values shall 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.

1.7.2.1 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; that is, members that are structs, unions, value types, sequences or arrays may not contain interface reference types. If the type of a member is a value type or contains a value type, it must meet all of the above constraints.

1.7.2.2 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 { };
};
```

1.7.3 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.

1.7.3.1 Factory operations

A factory operation is denoted by the **factory** keyword. A factory operation has a corresponding equivalent operation on the homeÕs 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 (Components::CreateFailure, <*exceptions*>);

A factory operation is required to support creation semantics; that is, the reference returned by the operation shall identify a component that did not exist prior to the operationÕs invocation. Factory operations are required to raise CreateFailure and may raise other exceptions.

1.7.3.2 Finder operations

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

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

The equivalent operation on the explicit interface is as follows:

***<component_type> <finder_operation_name>* (*<parameters>*)**
raises (Components::FinderFailure, <*exceptions*>);

A finder operation shall support the following semantics. The reference returned by the operation shall identify a previously-existing component managed by the home. The operation implementation determines which componentÕs reference to return based on the values of the operationÕs parameters. Finder operations are required to raise FinderFailure and may raise other exceptions.

1.7.3.3 Miscellaneous exports

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

1.7.4 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>
};

Given a derived home definition supporting one or more interfaces, as follows:

```
home <home_name> : <base_home_name>
        supports <interface_name>
        manages <component_type> {
    <explicit_operations>
};
```

The resulting explicit interface has the following form:

```
interface <home_name>Explicit : <base_home_name>Explicit,
<interface_name> {
    <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 1.7.3, ÒExplicit Operations in Home Definitions,Ó on page 1-37. 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 ***(HÕ, TÕ)*** or ***(HÕ, TÕ, K)***, where ***K*** is a primary key type specified on ***HÕ***, such that ***HÕ*** is derived from ***H***, then ***TÕ*** 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 ***(HÕ, TÕ)***, such that ***HÕ*** is derived from ***H***, then the definition of ***HÕ*** implicitly includes a primary key specification of type ***K***, becoming ***(HÕ, TÕ, K)***. The implicit interface for ***HÕ*** shall have the form specified for an implicit interface of a home with primary key ***K*** and component type ***TÕ***.

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 ***(HÕ, TÕ, KÕ)*** such that ***HÕ*** is derived from ***H***, then ***TÕ*** must be identical to or derived from ***T*** and ***KÕ*** 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 **CCMHome** are inherited by all home equivalent interfaces. These operations apply equally to homes with and without primary keys.

¥ Operations on **KeylessCCMHome** 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.

1.7.5 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 **CCMHome** and **KeylessCCMHome**.

¥ 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.

1.7.5.1 Orthodox operations

Because of the inheritance structure described in Section 1.7.4, ÒHome inheritance,Ó on page 1-38 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 1.7.2.1, ÒPrimary key type constraints,Ó on page 1-37) 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 than 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; that is, if the type of a member of the actual parameter value is a value type, only the state that constitutes the memberÕs 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 that 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 that are sequences or arrays are considered equal if all of their members are precisely equal, where order is significant.

1.7.5.2 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 ***(HÕ, TÕ, KÕ)***, such that ***HÕ*** is derived from ***H***, ***TÕ*** is derived from ***T***, and ***KÕ*** is derived from ***K***. The operation ***f*** (whose parameter type is ***K***) is inherited by equivalent interface for ***HÕ***. It may be the intended behavior of the designer that the actual type of the parameter to invocations of ***f*** on ***HÕ*** should be ***KÕ***, exploiting the polymorphism implied by inheritance of ***K*** by ***KÕ***. Alternatively, it may be the intended behavior of the designer that actual parameter values of either ***K*** or ***KÕ*** 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 home.

¥ The comparison behavior must meet the general definition of equivalence; that is, it must be symmetric, reflexive, and transitive.

1.7.6 CCMHome Interface

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

```
module Components {

    typedef unsigned long FailureReason;

    exception CreateFailure { FailureReason reason; };

    exception FinderFailure { FailureReason reason; };

    exception RemoveFailure { FailureReason reason; };

    exception DuplicateKeyValue { };

    exception InvalidKey { };

    exception UnknownKeyValue { };

    interface CCMHome {
        CORBA::IRObject get_component_def();
        CORBA::IRObject get_home_def ();
        void remove_component ( in CCMObject comp)
                raises (RemoveFailure);
    };
};
```

get_component_def

The **get_component_def** operation returns an object reference that supports the **CORBA::ComponentIR::ComponentDef** interface, describing the component type associated with the home object. In strongly typed languages, the **IRObject** returned must be narrowed to **CORBA::ComponentIR::ComponentDef** before use.

get_home_def

The **get_home_def** operation returns an object reference that supports the **CORBA::ComponentIR::HomeDef** interface describing the home type. In strongly typed languages, the **IRObject** returned must be narrowed to **CORBA::ComponentIR::HomeDef** before use.

remove_component

The **remove_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, **remove_component** raises a BAD_PARAM system exception. All other application errors raise the RemoveFailure exception.

Note Ð This specification does not define explicitly what the **FailureReason** values are for the CreateFailure, FinderFailure, and RemoveFailure exceptions. These values are currently vendor specific and will be standardized once consensus among vendors will be established.

1.7.7 KeylessCCMHome Interface

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

```
module Components {
    interface KeylessCCMHome {
        CCMObject create_component() raises (CreateFailure);
        };
};
```

create_component

The **create_component** operation creates a new instance of the component type associated with the home object. A home implementation may choose to disable the parameter-less **create_component** operation, in which case it shall raise a NO_IMPLEMENT system exception. All other failures raise the CreateFailure exception.

1.8 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 Ò**ComponentHomeFinder**Ó. This requires the following enhancement to the **ORB** interface definition:

```
module CORBA {

    interface ORB {
        Object resolve_initial_references (in ObjectID identifier)
            raises (InvalidName);
        };
};
```

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

```
module Components {

    exception HomeNotFound { };

    interface HomeFinder {
```

```
        CCMHome find_home_by_component_type (
            in CORBA::RepositoryId comp_repid)raises (HomeNotFound);
        CCMHome find_home_by_home_type (
            in CORBA::RepositoryId home_repid) raises (HomeNotFound);
        CCMHome 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 **comp_repid** 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 shall raise 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::KeylessCCMHome::create_component**. *The client must first attempt to narrow the* **CCMHome** *reference returned by the* **find_home_by_component_type** *to* **KeylessCCMHome**. *Otherwise, the client must have specific out-of-band knowledge regarding the home interface that may be returned, or the client must be sophisticated enough to obtain the* **HomeDef** *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 the type specified by the repository identifier in the **home_repid** parameter. If there are no homes of this type currently registered, the operation shall raise the HomeNotFound exception.

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

Since the operation returns a reference to **CCMHome**, 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 *Naming Service* specification (formal/01-02-65), section

2.4, ÒStringified Names.Ó The implementation of this operation may be delegated directly to an implementation of CORBA naming, 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 CORBA naming will do so for reasons of scalability and flexibility, in order, for example, to provide a home which is logically more ÒlocalÓ to the home finder (and thus, the client).*

¥ The clientÕs expectations regarding the returned reference, other than that it supports the **CCMHome** 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 the scope of 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 shall raise the HomeNotFound exception.

1.9 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 componentÕs 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 instanceÕs configuration. Component attributes are intended to be used during a component instanceÕs 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 supported interfaces. 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 componentÕs 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 assignÑin 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.*

1.9.1 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::CCMObject**) 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.

1.9.1.1 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 operation 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 3 - *CCM Implementation Framework*) and may not be disabled.

1.10 Configuration with Attributes

A componentÕs configuration is established primarily through its attributes. An *attribute configuration* is defined to be a description of a set of invocations on a componentÕs 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.

1.10.1 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.

1.10.1.1 The Configurator interface

The following interface is supported by all configurators:

```
module Components {

    exception WrongComponentType { };

    interface Configurator {
        void configure (in CCMObject 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 shall raise the WrongComponentType exception.

1.10.1.2 The StandardConfigurator interface

The **StandardConfigurator** has the following definition:

```
module Components {

    valuetype ConfigValue {
        public FeatureName name;
        public 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.

1.10.2 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 invoke **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 implementationÕs instance initialization mechanism (e.g., the component servant constructor) constitutes the default configuration.

1.10.2.1 HomeConfiguration interface

The implementation of a component typeÕs home object may optionally support the **HomeConfiguration** interface. The **HomeConfiguration** interface is derived from **Components::CCMHome**. 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 objectÕs 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 : CCMHome {
        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.

disable_home_configuration

This operation 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 invoke **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.

1.11 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 **CCMObject**.
- If a component type directly supports one or more IDL interfaces, the component interface is derived from both **CCMObject** and the supported interfaces.
- 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 **publishes** statements
- operations implied by **consumes** statements
- attributes

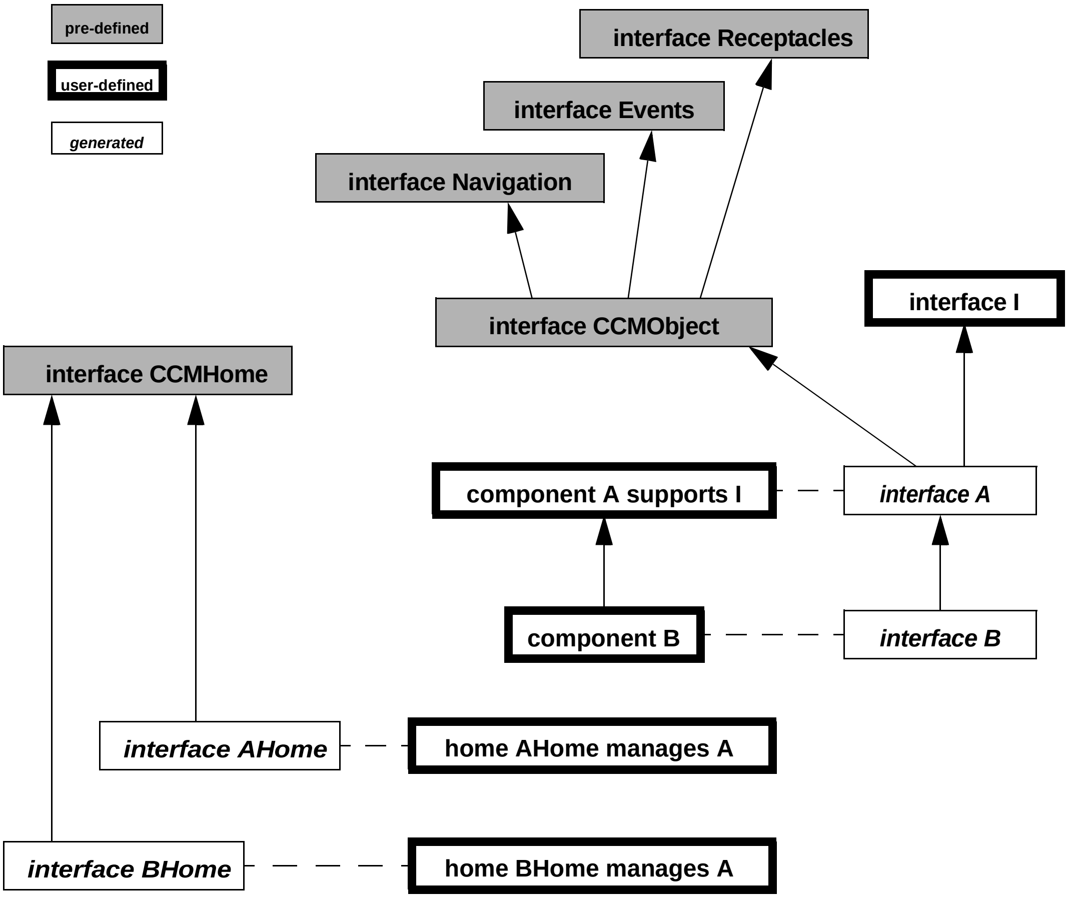


Figure 1-2 Component inheritance and related interface inheritance

1.11.1 CCMObject Interface

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

```
module Components {

    valuetype ComponentPortDescription
    {
        public FacetDescriptions facets;
        public ReceptacleDescriptions receptacles;
```

```
            public ConsumerDescriptions consumers;
            public EmitterDescriptions emitters;
            public PublisherDescriptions publishers;
        };

        exception NoKeyAvailable { };

        interface CCMObject : Navigation, Receptacles, Events {
            CORBA::IRObject get_component_def ( );
            CCMHome get_ccm_home( );
            PrimaryKeyBase get_primary_key( ) raises (NoKeyAvailable);
            void configuration_complete( ) raises (InvalidConfiguration);
            void remove() raises (RemoveFailure);
            ComponentPortDescription get_all_ports ();
        };

    };
```

get_component_def

This operation returns an **IRObject** reference to the component definition in the Interface Repository. The interface repository representation of a component is defined in the *Interface Repository Metamodel* chapter. In strongly typed languages, the **IRObject** returned must be narrowed to **CORBA::ComponentIR::ComponentDef** before use.

get_ccm_home

This operation returns a **CCMHome** reference to the home that manages this component.

get_primary_key

This operation is equivalent to the same operation on the componentÕs home interface. It returns a primary key value if the component is being managed by a home which defines a primary key. Otherwise, the NoKeyAvailable exception shall be raised.

configuration_complete

This operation is called by a configurator to indicate that the initial component configuration has completed. If the component determines that it is not sufficiently configured to allow normal client access, it raises the InvalidConfiguration exception. The component configuration process is described in Section 1.9, ÒComponent Configuration,Ó on page 1-45.

remove

This operation is used to delete a component. Application failures during **remove** may raise the RemoveFailure exception.

get_all_ports

The **get_all_ports** operation returns a value of type **ComponentPortDescription** containing information about all facets, receptacles, event sinks, emitted events and published events in the component's inheritance hierarchy. The order in which the information occurs in these sequences is not specified. If a component does not offer a port of any type, the associated sequence will have length zero.

1.12 Conformance Requirements

This section identifies the conformance points required for compliant implementations of the CORBA Component model.

The following conformance points are defined:

1. A CORBA COS vendor shall provide the relevant changes to the Lifecycle, Transaction, and Security Services identified in the following Section 1.12.2, ÒChanges to Object Services,Ó on page 1-55.
2. A CORBA ORB vendor need not provide implementations of Components aside from the changes made to the Core to support components. Conversely a CORBA Component vendor need not be a CORBA ORB vendor.
3. A CORBA Component vendor shall provide a conforming implementation of the Basic Level of CORBA Components.
4. A CORBA Component vendor may provide a conforming implementation of the Extended Level of CORBA Components.
5. To be conformant at the Basic level a non-Java product shall implement (at a minimum) the following:
   ¥ the IDL extensions and generation rules to support the client and server side component model for basic level components.
   ¥ CIDL. The multiple segment feature of CIDL (Section 2.12, ÒSegment Definition,Ó on page 2-10) need not be supported for basic components.
   ¥ a container for hosting basic level CORBA components.
   ¥ the XML deployment descriptors and associated zip files for basic components in the format defined in Section 6.1, ÒIntroduction,Ó on page 6-1.

   Such implementations shall work on a CORBA ORB as defined in #1 above.
6. To be conformant at the Basic level a Java product shall implement (at a minimum):
   ¥ EJB1.1, including support for the EJB 1.1 XML DTD.
   ¥ the java to IDL mapping, also known as RMI/IIOP.
   ¥ EJB to IDL mapping as defined in Section 5.3.2, ÒTranslation of CORBA Component requests into EJB requests,Ó on page 5-9.

   Such implementations shall work in a CORBA interoperable environment, including interoperable support for IIOP, CORBA transactions and CORBA security.

7. To be conformant at the extended level, a product shall implement (at a minimum) the requirements needed to achieve Basic PLUS:
   ¥ IDL extensions to support the client and server side component model for extended level components.
   ¥ A container for hosting extended level CORBA components.
   ¥ The XML deployment descriptors and associated zip files for basic and enhanced level components in the format defined in Section 6.1, ÒIntroduction,Ó on page 6-1.

   Such implementations shall work on a CORBA ORB as defined in #1 above.
8. A CORBA Component vendor may optionally support EJB clients interacting with CORBA Components, by implementing the IDL to EJB mapping as defined in Section 5.4.2, ÒTranslation of EJB requests into CORBA Component Requests,Ó on page 5-17.
9. This specification includes extensions to IDL, in the form of new keywords and grammar. Although a CORBA ORB vendor need not be a CORBA Component vendor, and vice-versa, it is important to maintain IDL as a single language. To this end, all compliant products of any conformance points above shall be able to parse any valid IDL definitions. However, it is permitted to raise errors, or to ignore, those parts of the grammar that relate to another conformance point.

Conforming implementations as defined above may also implement any additional features of this specification not required by the above conformance points.

1.12.1 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 CORBA Core, the *Interface Repository* chapter).

1.12.2 Changes to Object Services

1.12.2.1 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 **CosLifeCycle** with the following clarifications:

- The **factory_key** parameter is a name conforming to the Interoperable Naming Specification (orbos/98-10-11) for stringified names.
- The **factory_key** parameter is used as an input to the **find_home_by_name** operation on **Components::HomeFinder**.
- The default factory operation on the home is used to obtain a reference which can be narrowed to the **CosLifeCycle::GenericFactory** type.

1.12.2.2 Transaction Service

The following CORBA transaction service interface is changed to a local interface:

- **CosTransactions::Current**

1.12.2.3 Security Service

The following CORBA Security interfaces are changed to local interfaces:

- **SecurityLevel1::Current**
- **SecurityLevel2::PrincipalAuthenticator**
- **SecurityLevel2::Credentials**
- **SecurityLevel2::ReceivedCredentials**
- **SecurityLevel2::AuditChannel**
- **SecurityLevel2::AuditDecision**
- **SecurityLevel2::AccessDecision**
- **SecurityLevel2::QOPPolicy**
- **SecurityLevel2::MechanismPolicy**
- **SecurityLevel2::InvocationCredentialsPolicy**
- **SecurityLevel2::EstablishTrustPolicy**
- **SecurityLevel2::DelegationDirectivePolicy**
- **SecurityLevel2::Current**
- **SecurityReplacable::Vault**
- **SecurityReplacable::SecurityContext**
- **SecurityReplacable::ClientSecurityContext**
- **SecurityReplacable::ServerSecurityContext**

OMG CIDL Syntax and Semantics

2

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter describes OMG Component Implementation Definition Language (CIDL) semantics and gives the syntax for OMG CIDL grammatical constructs.

Contents

Note Ð Issue 5499

This chapter contains the following sections.

Section Title	**Page**
ÒOverviewÓ	2-2
ÒLexical ConventionsÓ	2-3
ÒOMG CIDL GrammarÓ	2-3
ÒOMG CIDL SpecificationÓ	2-5
ÒComposition DefinitionÓ	2-6
~~ÒCatalog Usage DeclarationÓ~~	~~2-7~~
ÒHome Executor DefinitionÓ	2-8
ÒHome Implementation DeclarationÓ	2-9
ÒStorage Home BindingÓ	2-9
ÒHome Persistence DeclarationÓ	2-10
ÒExecutor DefinitionÓ	2-10

Section Title	Page
ÒSegment DefinitionÓ	2-10
ÒSegment Persistence DeclarationÓ	2-11
ÒFacet DeclarationÓ	2-12
ÒFeature Delegation SpecificationÓ	2-12
ÒAbstract Storage Home Delegation SpecificationÓ	2-13
ÒExecutor Delegation SpecificationÓ	2-15
ÒAbstract Spec DeclarationÓ	2-16
ÒProxy Home DeclarationÓ	2-16

2.1 Overview

The OMG Component Implementation Definition Language (CIDL) is a language used to describe the structure and state of component implementations. Component-enable ORB products generate implementation skeletons from CIDL definitions. Component builders extend these skeletons to create complete implementations.

OMG CIDL obeys the same lexical rules as OMG Persistent State Definition Language (PSDL) and OMG IDL, although new keywords are introduced to support concepts specific to component implementation descriptions.

The description of OMG CIDLÕs lexical conventions is presented in Section 2.2, ÒLexical Conventions,Ó on page 2-3. A description of OMG IDL preprocessing is presented in CORBA Core, IDL Syntax and Semantics chapter, Preprocessing section. The scope rules for identifiers in an OMG IDL specification are described in CORBA Core, IDL Syntax and Semantics chapter, CORBA Module section.

The OMG CIDL grammar is an extension of a combination of the OMG PSDL and OMG IDL grammars, with new constructs to define component implementations. OMG CIDL is a declarative language. The grammar is presented in Section 2.3, ÒOMG CIDL Grammar,Ó on page 2-3.

A source file containing specifications written in OMG CIDL must have a Ò.cdlÓ extension.

The description of OMG CIDL grammar uses the same syntax notation that is used to describe OMG IDL in CORBA Core, *IDL Syntax and Semantics* chapter. For reference, Table 2-1 lists the symbols used in this format and their meaning.

Table 2-1 IDL EBNF

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal

Table 2-1 IDL EBNF *(Continued)*

Symbol	Meaning
ÒtextÓ	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optionalÑmay occur zero or one time

2.2 Lexical Conventions

This section presents the lexical conventions of OMG CIDL. In general OMG CIDL uses the same lexical conventions as OMG PSDL and OMG IDL. It does use additional keywords as described below.

2.2.1 Keywords

Note Ð Issue 5499

The identifiers listed in Table 2-2 are reserved for use as keywords in CIDL, and may not be used otherwise in CIDL, unless escaped with a leading underscore**.** These are in addition to the ones defined by PSDL and IDL, which may also not be used otherwise in CIDL, unless escaped with a leading underscore**.**

Table 2-2 Keywords

bindsTo	delegatesTo	facet	proxy	session
~~catalog~~	entity	implements	segment	storage~~H~~home
composition	executor	process	service	storedOn

2.3 OMG CIDL Grammar

The CIDL grammar is a combination of the PSDL and IDL grammars plus the following productions:

Note Ð Issues 5498, 5499, and 5500

(1) <cidl_specification> ::= <import>* <cidl_definition>+

(2) <cidl_definition> ::= <type_dcl> Ò;Ó
| <const_dcl> Ò;Ó
| <except_dcl> Ò;Ó
| <interface> Ò;Ó
| <cidl_module> Ò;Ó
| <storage~~_~~home> Ò;Ó
| <abstract_storagehome> Ò;Ó
| <storagetype> Ò;Ó
| <abstract_storagetype> Ò;Ó

| <value> Ò;Ó
| <type_id_dcl> Ò;Ó
| <type_prefix_dcl> Ò;Ó
| <event> Ò;Ó
| <component> Ò;Ó
| <home_dcl> Ò;Ó
| <composition> Ò;Ó

(3) <cidl_module> ::= ÒmoduleÓ <identifier>
Ò{Ó <cidl_definition>+ Ò}Ó

(4) <composition> ::= ÒcompositionÓ <category> <identifier>
Ò{Ó <composition_body> Ò}Ó

(5) <category> ::= ÒentityÓ
| ÒprocessÓ
| ÒserviceÓ
| ÒsessionÓ

(6) <composition_body> ::= ~~[<catalog_use_dcl>]~~ <home_executor_def>
[<proxy_home_def>]

~~(7) <catalog_use_dcl> ::= ÒusesÓ ÒcatalogÓ Ò{Ò <catalog_dcl>+ Ò}Ó Ò;Ó~~

~~(8) <catalog_dcl> ::= <catalog_type_spec> <catalog_label>~~

~~(9) <catalog_type_spec> ::= <scoped_name>~~

~~(10) <catalog_label> ::= <identifier>~~

(7) <home_executor_def> ::= ÒhomeÓ ÒexecutorÓ <identifier> Ò{Ó
<home_executor_body> Ò}Ó Ò;Ó

(8) <home_executor_body> ::= <home_impl_dcl>
[<abstract_storage_home_binding>]
[<~~stored_on~~home_persistence_dcl>]
<executor_def>
[<abstract_storage_home_delegation_spec>]
[<executor_delegation_spec>]
[<abstract_spec>]

(9) <home_impl_dcl> ::= ÒimplementsÓ <home_type_name> Ò;Ó

(10) <home_type_name> ::= <scoped_name>

(11)<abstract_storage_home_binding>::= ÒbindsToÓ
<abstract_storage_home_name> Ò;Ó

~~(16)<abstract_storage_home_name>::= <catalog_label> Ò.Ó <abstract_storage_home_label>~~

~~(17)<abstract_storage_home_label>::= <identifier>~~

(12) <home_persistence_dcl> ::= ÒstoredOnÓ
<abstract_storage_home_name> Ò;Ó

(13) <executor_def> ::= ÒmanagesÓ <identifier>
[<executor_body>] Ò;Ó

(14) <executor_body> ::= Ò{Ó <executor_member>+ Ò}Ó

(15) <executor_member> ::= <segment_def>
| <feature_delegation_spec>

(16) <segment_def> ::= ÒsegmentÓ <identifier>
Ò{Ó <segment_member>+ Ò}Ó

```
(17)        <segment_member>   ::=  <segment_persistence_dcl> Ò;Ó
                               |    <facet_dcl> Ò;Ó
(18)<segment_persistence_dcl>::=  ÒstoredOnÓ
                                    <abstract_storage_home_name> Ò;Ó
(19)               <facet_dcl>  ::=  ÒprovidesÓ ÒfacetÓ <identifier>
                                     { Ò,Ó <identifier> }*
(20)<feature_delegation_spec> ::=ÒdelegatesToÓ ÒstorageÓ
                                     <feature_delegation_list>
(21) <feature_delegation_list>  ::=  Ò(Ó <feature_delegation> { Ò,Ó
                                    <feature_delegation> }* Ò)Ó
(22)      <feature_delegation>  ::=  <feature_name> Ò:Ó
                                     <storage_member_name>
(23)            <feature_name>  ::=  <identifier>
(24)<storage_member_name>       ::=  <identifier>
(25)<abstract_storage_home_delegation_spec>::= ÒdelegatesToÓ ÒabstractÓ
                                     ÒstoragehomeÓ <delegation_list> Ò;Ó
(26)<executor_delegation_spec> ::= ÒdelegatesToÓ ÒexecutorÓ
                                    <delegation_list> Ò;Ó
(27)          <delegation_list>  ::=  Ò(Ó <delegation> { Ò,Ó <delegation> }* Ò)Ó
(28)              <delegation>  ::=  <operation_name> [ Ò:Ó <operation_name> ]
(29)          <operation_name>  ::=  <identifier>
(30)           <abstract_spec>  ::=  ÒabstractÓ <operation_list> Ò;Ó
(31)          <operation_list>  ::=  Ò(Ó <operation_name>
                                     { Ò,Ó <operation_name> }* Ò)Ó
(32)          <proxy_home_def>  ::= ÒproxyÓ ÒhomeÓ <identifier>
                                    Ò{Ó <proxy_home_member>+ Ò}Ó Ò;Ó
(33)   <proxy_home_member>  ::= <home_delegation_spec> Ò;Ó
                               |     <abstract_spec>
(34) <home_delegation_spec>  ::=  ÒdelegatesToÓ ÒhomeÓ <delegation_list>
```

2.4 OMG CIDL Specification

A CIDL specification is like a PSDL and IDL specification that could also contain composition definitions. The syntax is:

Note Ð Issue 5498

```
(1)       <cidl_specification>  ::=  <import>* <cidl_definition>+
(2)          <cidl_definition>  ::=  <type_dcl> Ò;Ó
                               |    <const_dcl> Ò;Ó
                               |    <except_dcl> Ò;Ó
                               |    <interface> Ò;Ó
                               |    <cidl_module> Ò;Ó
                               |    <storage_home> Ò;Ó
                               |    <abstract_storagehome> Ò;Ó
                               |    <storagetype> Ò;Ó
```

| <abstract_storagetype> Ò;Ó
| <value> Ò;Ó
| <type_id_dcl> Ò;Ó
| <type_prefix_dcl> Ò;Ó
| <event> Ò;Ó
| <component> Ò;Ó
| <home_dcl> Ò;Ó
| <composition> Ò;Ó
(3) <cidl_module> ::= ÒmoduleÓ <identifier>
Ò{Ó <cidl_definition>+ Ò}Ó

2.5 Composition Definition

The syntax for composition definitions is as follows:

Note Ð Issues 5498 and 5499

(4) <composition> ::= ÒcompositionÓ <category> <identifier> Ò{Ó
<composition_body> Ò}Ó
(5) <category> ::= ÒentityÓ
| ÒprocessÓ
| ÒserviceÓ
| ÒsessionÓ
(6) <composition_body> ::= ~~[<catalog_use_dcl>]~~ <home_executor_def>
[<proxy_home_def>]

A composition definition is a named scope that contains elements that constitute the composition. The elements of a composition definitions are as follows:

- The keyword **composition**.
- The specification of the life cycle category, one of the keywords **service**, **session**, **process**, or **entity**. Subsequent definitions and declarations in the composition must be consistent with the declared category, as defined in Section 2.5.1, ÒLife Cycle Category and Constraints,Ó on page 2-6.
- An identifier that names the composition in the enclosing module scope.
- The composition body.

The composition body consists of the following elements:

- ~~an optional catalog usage declaration,~~
- a mandatory home executor definition, and
- an optional proxy home definition.

2.5.1 Life Cycle Category and Constraints

Certain composition configurations are only valid for certain life cycle categories. The Container Programming Model chapter describes the life cycle-related constraints from the perspective of the container. These constraints map onto corresponding constraints in component and composition definitions. The following lists define the CIDL constructs that are either mandatory or invalid for the designated life cycle category.

Note that these constraints supersede the conditionality of constructs based on CIDL syntax. If a construct is described below as mandatory for the category in question, it is mandatory regardless of its syntactic properties. All of the constructs described as invalid for a particular category are, of necessity, syntactically optional.

Table 2-3 Constraints for service and session components

Service and Session	**Mandatory**	**None**
	Invalid	Abstract storage home bound to home executor: **<abstract_storage~~-~~_home_binding>** in home executor body.
		Component home implemented by home executor specifies a primary key.
		Component home implemented by home executor specifies explicit finder operations.
		Segmented executor: **<segment_def>** in executor body.

Table 2-4 Constraints for process components

Process	**Mandatory**	**None**
	Invalid	Component home implemented by home executor specifies a primary key.

Table 2-5 Constraints for entity components

Entity	**Mandatory**	Component home implemented by home executor specifies a primary key.
	Invalid	none

Note Ð Issue 5499

~~2.6 Catalog Usage Declaration~~

~~The syntax for a catalog usage declaration is as follows:~~

~~**(7) <catalog_use_dcl> ::= ÒusesÓ ÒcatalogÓ Ò{Ò <catalog_dcl>+ Ò}Ó Ò;Ó**~~
~~**(8) <catalog_dcl> ::= <catalog_type_spec> <catalog_label>**~~
~~**(9) <catalog_type_spec> ::= <scoped_name>**~~
~~**(10) <catalog_label> ::= <identifier>**~~

~~A catalog usage declaration consists of the following elements:~~

¥ ~~the keywords **uses** and **catalog**, and~~
¥ ~~a block containing one or more catalog label declarations.~~

~~A catalog label declaration consists of the following elements:~~

¥ ~~a scoped name denoting a previously defined catalog, and~~
¥ ~~an identifier that denotes a putative catalog of the specified type within the scope of the composition.~~

~~A catalog usage declaration identifies catalog types that are used by the composition and assigns them labels that are used within the scope of the composition to refer to a putative catalog of the specified type. A catalog usage declaration also causes the CIF to generate implementation of the following behaviors.~~

~~During the activation of a home executor, the CIF-generated activate implementation on the home executor shall obtain the **CosPersistentState::CatalogBase** interface from the component context, and invoke **get_catalog** on it, requesting a catalog of each type specified in the catalog usage declaration. The catalogs are requested by their repository ID values. The home shall maintain references to the specified catalogs, and make them available to the executors.~~

2.7 Home Executor Definition

The syntax for a home executor definition is as follows:

Note Ð Issues 5498 and 5500

(7) <home_executor_def> ::= ÒhomeÓ ÒexecutorÓ <identifier>
Ò{Ó <home_executor_body> Ò}Ó Ò;Ó
(8) <home_executor_body> ::= <home_impl_dcl>
[<abstract_storage~~_~~home_binding>]
[<~~stored_on~~home_persistence_dcl>]
<executor_def>
[<abstract_storage~~_~~home_delegation_spec>]
[<executor_delegation_spec>]
[<abstract_spec>]

A home executor definition consists of the following elements:

¥ the keywords **home** and **executor**,
¥ an identifier that names the home executor definition within the scope of the composition, and
¥ a home executor body.

The home executor body consists of the following elements:

- A home implementation declaration.
- An optional abstract storage home binding, specifying the storage home upon which the components managed by the home are stored.
- An optional home persistence declaration, identifying an abstract storage home upon which the state of the home executor itself is to be stored.
- An executor definition, describing the component executor managed by the home executor.
- An optional delegation specification describing the mapping of home operations to storage home operations.
- An optional delegation specification describing the mapping of home factory operations to the operations on the component executor.
- An optional abstract specification, declaring operations on the home executor that are to be left unimplemented, overriding default generated implementations.

The <*identifier*> in the header of the home executor definition is used as the basis for the name of the skeleton artifact generated by the CIF. The specific forms of the executors are defined in language mappings. The general requirements for language mappings of home executors are defined in Section 2.7, ÒHome Executor Definition,Ó on page 2-8.

2.8 Home Implementation Declaration

The syntax of a home implementation declaration is as follows:

Note Ð Issue 5498

(9) <home_impl_dcl> ::= ÒimplementsÓ <home_type_name> Ò;Ó
(10) <home_type_name> ::= <scoped_name>

The home implementation declaration consists of the following elements:

- the keyword **implements**, and
- a scoped name denoting a component home imported from IDL.

The home implementation declaration specifies the component home that is to be implemented by the home executor being defined. The generated skeleton must support the home equivalent interface, as defined in Section 1.7.1, ÒEquivalent Interfaces,Ó on page 1-33. Implementations of orthodox home operations are generated if the life cycle category of the composition is either **entity** or **process** and the home executor specifies an abstract storage home binding, or if the life cycle category of the executor is either **session** or **service**.

The detailed semantics of generated implementations are described in Section 2.8, ÒHome Implementation Declaration,Ó on page 2-9.

2.9 Storage Home Binding

The syntax for a storage home binding is as follows:

Note Ð Issues 5498, 5499, and 5500

(11)<abstract_storage~~_~~home_binding>::= ÒbindsToÓ
<abstract_storage~~_~~home_name> Ò;Ó
~~**(56)<abstract_storage_home_name>::= <catalog_label> Ò.Ó**~~
~~**<abstract_storage_home_label>**~~
~~**(57)<abstract_storage_home_label>::= <identifier>**~~

An abstract storage home binding declaration consists of the following elements:

¥ the keyword **bindsTo**, and
¥ an abstract storage home name.

Note Ð Issue 5499

~~An abstract storage home name consists of a catalog label, a period separator, and a storage home label. The catalog label must denote a catalog previously declared in the catalog usage declaration in the current composition definition. The storage home label must denote a storage home declared as a member of the catalog type associated with the catalog label.~~

2.10 Home Persistence Declaration

The syntax for a home persistence declaration is as follows:

Note Ð Issues 5498 and 5500

(12) <home_persistence_dcl> ::= ÒstoredOnÓ
<abstract_storage~~_~~home_name> Ò;Ó

A home persistence declaration consists of the following elements:

¥ the keyword **storedOn**, and
¥ an abstract storage home name.

A home persistence declaration establishes that the home executor is itself persistent, and that its persistent state is managed by the container. The abstract storage type of the specified abstract storage home constitutes the state of the component home. The specific responsibilities of generated home executors related to home persistence are described in Section 2.9, ÒStorage Home Binding,Ó on page 2-9.

2.11 Executor Definition

The syntax for an executor definition is as follows:

Note Ð Issue 5498

```
(13)          <executor_def>   ::=  ÒmanagesÓ <identifier>
                                    [ <executor_body> ] Ò;Ó
(14)         <executor_body>   ::=  Ò{Ó <executor_member>+ Ò}Ó
(15)       <executor_member>   ::=  <segment_def>
                               |     <feature_delegation_spec>
```

An executor definition has the following elements:

¥ the keyword **manages**,
¥ and identifier that names the component executor being defined, and
¥ an executor body, containing one or more members enclosed in braces.

An executor member is either a *segment definition* or a *feature delegation specification*, as defined below.

The identifier in the executor definition forms the basis of the name of the programming artifact generated as the executor skeleton. The details of executor structure and responsibilities are defined in Section 2.7, ÒHome Executor Definition,Ó on page 2-8, and in CIDL language mappings.

2.12 Segment Definition

The syntax for a segment definition is as follows:

Note Ð Issue 5498

```
(16)           <segment_def>   ::=  ÒsegmentÓ <identifier>
                                    Ò{Ó <segment_member>+ Ò}Ó
(17)        <segment_member>   ::=  <segment_persistence_dcl> Ò;Ó
                               |    <facet_dcl> Ò;Ó
```

A segment definition consists of the following elements:

¥ the keyword **segment**,
¥ an identifier that names the segment in the scope of the executor definition, and
¥ one or more segment members enclosed in braces.

A segment member is either a *segment persistence declaration*, or a *facet declaration*, as described below.

If a segment definition occurs in an executor definition, the corresponding executor is said to be a segmented executor. If no segment definition occurs in an executor definition, the executor is said to be monolithic.

A separate skeleton is generated by the CIF for each segment of a segmented executor. Segments are independently activated. Each segment is assigned a segment identifier, which as a numeric value of type short, by the CIF implementation. The segment identifier is interpreted internally by the generated implementation during activation. Segment identifiers are also used in component identities, as described in Section 4.4.3.1, ÒComponent Identifiers,Ó on page 4-38. There is no canonical mechanism for assigning segment identifier values (other than the component segment), as the values of segment identifiers does not affect portability or interoperability.

All executors have a distinguished segment, the component segment, that supports the component facet (i.e., the facet supporting the component equivalent interface). The segment identifier value of the component segment is always zero. If a component does not explicitly declare segments, the monolithic executor is still considered in some contexts to be the component segment executor.

The details of segment structure and implementation responsibilities are described in Section 2.12, ÒSegment Definition,Ó on page 2-10.

2.13 Segment Persistence Declaration

The syntax for a segment persistence declaration is as follows:

Note Ð Issues 5498 and 5500

---

(18)<segment_persistence_dcl>::= ÒstoredOnÓ
<abstract_storage~~-~~_home_name> Ò;Ó

A segment persistence declaration has the following elements:

- the keyword **storedOn**, and
- an abstract storage home name.

A segment persistence declaration specifies the abstract storage home upon which the state of the segment will be stored. The abstract storage type of the storage home constitutes the state of the segment.

The detailed structure of segments, and implementation responsibilities with respect to segment persistence are described in Section 2.13, ÒSegment Persistence Declaration,Ó on page 2-11.

2.14 Facet Declaration

The syntax for a facet declaration is as follows:

Note Ð Issue 5498

---

(19) <facet_dcl> ::= ÒprovidesÓ ÒfacetÓ <identifier>
{ Ò,Ó <identifier> }*

A facet declaration has the following elements:

- The keywords **provides** and **facet**.

¥ One or more identifiers separated by commas, where each identifier denotes a facet defined by the component type implemented by the composition (i.e., the component type managed by the home that is implemented by the home executor defined in the composition).

A facet declaration associates one or more component facets with the segment. The generated segment executor will provide the specified facets. A facet name may only appear in a single segment definition. Facets that are not explicitly declared in a segment definition are provided by the component segment.

The detailed structure of segments, and implementation responsibilities with respect to providing facets are described in Section 2.14, ÒFacet Declaration,Ó on page 2-12.

2.15 Feature Delegation Specification

The syntax for a feature delegation specification is as follows:

Note Ð Issue 5498

```
(20)<feature_delegation_spec> ::=ÒdelegatesToÓ ÒstorageÓ
                                   <feature_delegation_list>
(21) <feature_delegation_list>   ::=  Ò(Ó <feature_delegation> { Ò,Ó
                                   <feature_delegation> }* Ò)Ó
(22)       <feature_delegation>   ::=  <feature_name> Ò:Ó
                                   <storage_member_name>
(23)              <feature_name>   ::=  <identifier>
(24)<storage_member_name>   ::=  <identifier>
```

A feature delegation specification has the following elements:

¥ the keywords **delegatesTo**, **abstract** and **storagetype**, and

¥ a list of feature delegation specifications, enclosed in parentheses and separated by commas.

A feature delegation specification consists of the following elements:

¥ An identifier that denotes a stateful feature of the component implemented by the composition.

¥ A colon.

¥ An identifier that denotes a member of the abstract storage type of the abstract storage home specified in the abstract storage home binding in the home executor definition.

A feature delegation specification defines an association between a stateful feature of the component being implemented and a member of the abstract storage type that incarnates the component (or the component segment). The component executor skeleton generated by the CIF will provide implementations of feature management operations that store the featureÕs state in the specified storage member. Stateful features include attributes, receptacles, and event sources.

The following constraints regarding feature delegation must be observed:

¥ Feature delegation specifications may only occur in an executor definition when the home executor specified an abstract storage home binding.

¥ The type of the storage member specified in a feature delegation must be compatible with the type of the feature. Compatibility, for the purposes of feature delegation is defined in Table 2-6.

Table 2-6 Type compatibility for feature delegation purposes

Feature	**Storage member type**
attribute	Must be identical to feature for all types except object reference and valuetype; for object reference and valuetype storage member must be of identical type or base type (direct or indirect).
receptacle (simplex)	Must be identical to feature type or base interface (direct or indirect) of feature type.
receptacle (multiplex)	Sequence of type compatible with receptacle type as defined above.
emitter event source	Must be identical to feature type or base interface (direct or indirect) of feature type.
publisher event source	long*

* The persistent state maintained internally by the component is the **ChannelId** of the notification channel created by the container.

2.16 *Abstract Storage Home Delegation Specification*

The syntax for a storage home delegation specification is as follows:

Note Ð Issue 5498

```
(25)<abstract_storage_home_delegation_spec>::= ÒdelegatesToÓ ÒabstractÓ
                                               ÒstoragehomeÓ <delegation_list> Ò;Ó
(27)          <delegation_list>   ::=  Ò(Ó <delegation> { Ò,Ó <delegation> }* Ò)Ó
(28)               <delegation>   ::=  <operation_name> [ Ò:Ó <operation_name> ]
(29)           <operation_name>   ::=  <identifier>
```

An abstract storage home delegation specification has the following elements:

- ¥ The keywords **delegatesTo**, **abstract**, and **storagehome**.
- ¥ A list of delegation specifications enclosed in parentheses and separated by commas.

A delegation specification has the following elements:

- ¥ An identifier that denotes an operation on the home equivalent interface supported by the home executor.

  - An optional delegation target, consisting of a colon, followed by identifier that denotes an operation on the abstract storage home to which the home is bound (i.e., the abstract storage home specified in the abstract storage home binding).

An abstract storage home delegation specification associates an operation on the home interface with an operation on the abstract storage home interface. The CIF shall generate an implementation of the specified home operation that delegates to the specified abstract storage home operation.

If the optional delegation target is omitted, the home operation is assumed to be delegated to an operation on the abstract storage home with the same name. If no such operation exists on the abstract storage home, the specification is not legal.

The signature of the abstract storage home operation must be compatible with the abstract storage home. Signature compatibility, from the perspective of abstract storage home delegation, has the following definition:

- If the home operation is an explicit **factory** operation, the abstract storage home operation must be an explicit **factory** operation.
- If the home operation is not a factory, the return type of the home operation must be identical to the return type of the abstract storage home operation, except when the return type is an object reference type or a value type. If the return type of the home operation is an object reference type or a value type, the return type of the storage home operation must be identical to, or more derived than, the return type of the home operation.
- For each exception explicitly raised by the storage home operation, an identical exception must appear in the **raises** clause of the home operation. The inverse is not trueÑthe home operation may raise exceptions not raised by the abstract storage home operation.
- The number of parameters in the parameter lists of the home operation and the abstract storage home operation must be equal. Each parameter in the abstract storage home operation must be compatible with the parameter in the same position in the signature of the home operation, where compatibility is defined as follows:
  - If the parameter in the home operation is neither an object reference type nor a value type, the type of the corresponding parameter in the abstract storage home operation must be identical.
  - If the parameter type in the home operation is an object reference and the parameter is an **in** parameter, the corresponding parameter in the abstract storage home operation must be identical to, or a base type (direct or indirect) of, the parameter in the home operation.
  - If the parameter type in the home operation is an object reference and the parameter is an **out** parameter, the corresponding parameter in the abstract storage home operation must be identical to, or more derived than, the parameter in the home operation.
  - If the parameter type in the home operation is an object reference and the parameter is an **inout** parameter, the corresponding parameter in the abstract storage home operation must be identical to the parameter in the home operation.

The following additional constraints and rules apply to abstract storage home delegation:

- An operation on the home interface may delegate to at most one operation on the abstract storage home interface.
- An operation on the abstract storage home interface may be the target of at most one delegation from the home interface.
- Implicitly defined operations on the home (i.e., orthodox operations) delegate by default to cognate operations on the abstract storage home, as described by Section 1.7.5.1, ÒOrthodox operations,Ó on page 1-40. These default delegations may be over-ridden by explicit delegations. If an operation on the abstract storage home that is normally the default target of a delegation appears as the target of an explicit delegation, then the home operation that normally would have delegated to that target by default shall have no generated implementation (unless one is explicitly defined).

The detailed semantics and implementation responsibilities of delegated abstract storage home operations are described in Section 2.16, ÒAbstract Storage Home Delegation Specification,Ó on page 2-13.

2.17 Executor Delegation Specification

The syntax for an executor delegation specification has the following form:

Note Ð Issue 5498

(26)<executor_delegation_spec> ::= ÒdelegatesToÓ ÒexecutorÓ
<delegation_list> Ò;Ó

An executor delegation specification consists of the following elements:

- the keywords **delegatesTo** and **executor**, and
- a delegation list, identical structurally to the delegation list of the abstract storage home delegation specification.

An executor delegation specification defines an operation on the component executor, to which the specified home operation will be delegated. The following constraints apply to executor delegation specifications:

- Only factory operations may be delegated to the executor, including explicitly declared factories and implicit create operations.
- If no delegation target is explicitly specified, the operation defined on the executor shall have the same name as the delegating home operation.
- The signature of the defined operation on the executor shall be identical to the signature of the home operation, with the exception that the return type of the executor operation shall be void if the home does not specify a primary key, or the return type shall be the type of the primary key if the home specifies a primary key.

The CIF shall generate an implementation of the home operation that delegates to the defined operation on the executor. The detailed semantics and implementation responsibilities are described in Section 2.17, ÒExecutor Delegation Specification,Ó on page 2-15.

2.18 Abstract Spec Declaration

The syntax for an abstract spec has the following form:

Note Ð Issue 5498

```
(30)          <abstract_spec>   ::=  ÒabstractÓ <operation_list> Ò;Ó
(31)          <operation_list>  ::=  Ò(Ó <operation_name>
                                      { Ò,Ó <operation_name> }* Ò)Ó
```

2.19 Proxy Home Declaration

The syntax for a proxy home declaration has the following form:

```
(32)         <proxy_home_def>   ::= ÒproxyÓ ÒhomeÓ <identifier>
                                     Ò{Ó <proxy_home_member>+ Ò}Ó Ò;Ó
(33)   <proxy_home_member>   ::= <home_delegation_spec> Ò;Ó
                             |      <abstract_spec>
(34) <home_delegation_spec>  ::=  ÒdelegatesToÓ ÒhomeÓ <delegation_list>
```

CCM Implementation Framework 3

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter describes the semantics of the CORBA Component Model Implementation Framework.

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒIntroductionÓ	3-1
ÒComponent Implementation Framework (CIF) ArchitectureÓ	3-2
ÒLanguage MappingÓ	3-38

3.1 Introduction

The Component Implementation Framework (CIF) defines the programming model for constructing component implementations. Implementations of components and component homes are described in CIDL. See the ÒOMG CIDL Syntax and Semantics" chapter for the definition and syntax. The CIF uses CIDL descriptions to generate programming skeletons that automate many of the basic behaviors of components, including navigation, identity inquiries, activation, state management, lifecycle management, and so on.

3.2 Component Implementation Framework (CIF) Architecture

As a programming abstraction, the CIF is designed to be compatible with the existing POA framework, but also to insulate programmers from its complexity. In particular, the CIF can be implemented using the existing POA framework, but it does not directly expose any elements of that framework.

3.2.1 Component Implementation Definition Language (CIDL)

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.

3.2.2 Component persistence and behavior

CIDL is a superset of the Persistent State Definition Language, defined in the Persistent State Service specification (document orbos/99-07-07).

A CIDL implementation definition may optionally associate an abstract storage type with the component implementation, such that the abstract storage type defines the form of the internal state encapsulated by the component. When a component implementation declares an associated abstract storage type in this manner, the CIF and the run-time container environment cooperate to manage the persistence of the component state automatically.

This chapter addresses the elements of the CIF that pertain to the implementation of a componentÕs behavior.

3.2.3 Implementing a CORBA Component

The remainder of Section 3.2 provides an overview of the concepts involved in building component implementations. It is intended to provide a high-level description that will serve as a framework for understanding the more formal descriptions that follow in subsequent sections. While the information in this section is normative (with the exception of italicized, indented rationale), it is not intended to be a complete or precise specification of the CIF, or all of the possible design options from which a component implementor may choose.

3.2.4 Behavioral elements: Executors

We coin the term *executor* to indicate the programming artifact that supplies the behavior of a component or a component home. In general, the terms *executor* or *component executor* refer to the artifact that implements the component type, and the term *home executor* refers to the artifact that implements the component home.

> *We chose to use the word* executor *rather than* servant *to avoid confusion with POA servants. POA servants, while conceptually similar to executors,*

are significantly different in detail, and map to different types in programming languages. Executor is pronounced with the accent on the second syllable (e.g.-ZEK-yoo-tor).

We have tried to avoid terminology that is specific to object-oriented programming languages, such as class, base class, derive, and so on, in an attempt to be precise and acknowledge that the CIF framework may be mapped to procedural programming languages. Hence, we typically use the word artifact *or* programming artifact *to denote what may conveniently be thought of as a class, and likewise, the term* skeleton *to denote a generated abstract base class that is extended to form a complete implementation class. We hope this is not overly distracting to the reader.*

3.2.5 Unit of implementation : Composition

An implementation of a component comprises a potentially complex set of artifacts that must exhibit specific relationships and behaviors in order to provide a proper implementation. The CIDL description of a component implementation is actually a description of this aggregate entity, of which the component itself may be a relatively small part. In order to enable more concise discussion, we coin the term *composition* to denote both the set of artifacts that constitute the unit of component implementation, and the definition itself. **composition** is the CIDL meta-type that corresponds to an implementation definition.

A composition definition specifies the following elements:

Component home

A composition definition specifies a component home type, imported from IDL. The specification of a component home implicitly identifies the component type for which the composition provides an implementation (i.e., the component type managed by the home, as specified in the IDL home definition).

Abstract Storage home binding

A composition optionally specifies an abstract storage home to which the component home is bound. The specification of an abstract storage home binding implicitly identifies the abstract storage type that incarnates the component. The relationship **between a home and the component it manages to isomorphic to the relationship** between an abstract storage home and the abstract storage type it manages. When a home binds to an abstract storage home, the component managed by the home is implicitly bound to the abstract storage type of this abstract storage home.

Home executor

A composition definition specifies a home executor definition. The name of the home executor definition is used as the name of the programming artifact (e.g., the class) generated by the CIF as the skeleton for the home executor. The contents of the home executor definition describe the relationships between the home executor and other elements of the composition, determining the characteristics of the generated home executor skeleton.

Component executor

A composition specifies an executor definition. The name of the executor definition is used as the name of the programming artifact generated by the CIF as the skeleton of the component executor. The body of the executor definition optionally specifies executor *segments*, which are physical partitions of the executor, encapsulating independent state and capable of being independently activated. Segments are described in Section 3.2.9.1, ÒSegmented executors,Ó on page 3-27. The executor body may also specify a mapping, or *delegation*, of certain component features (e.g., attributes) to storage members.

Delegation specification

A composition may optionally provide a specification of home operation delegation. This specification maps operations defined on the component home to isomorphic operations on either the abstract storage home or the component executor. The CIF uses this description to generate implementations of operations on the home executor, and to generate operation declarations on the component executor.

Proxy home

A composition may optionally specify a proxy home. The CIF supports the ability to define proxy home implementations, which are not required to be collocated with the container that executes the component implementation managed by the home. In some configurations, proxy homes can provide implementations of home operations without contacting the container that executes the actual home and component implementation. Support for proxy homes is intended to increase the scalability of the CORBA Component Model. The use of proxy homes is completely transparent to component clients and, to a great extent, transparent to component implementations. Proxy home behavior is described in Section 3.2.10.1, ÒProxy home delegation,Ó on page 3-36.

3.2.6 Composition structure

A composition binds all of the previously-described elements together, and requires that the relationships between the bound entities define a consistent whole.

Note that a component home type necessarily implies a component type; that is, the managed component type specified in the home definition. Likewise, an abstract storage home implies an abstract storage type. It is unnecessary, therefore, for a composition to explicitly specify a component type or an abstract storage type. They are implicitly determined by the specification of a home and abstract storage home.

> *It may seem odd that the center of focus for compositions is the home rather than the component, but this works out to be reasonably intuitive in practice. The home is the primary point of contact for a client, and the homeÕs interface and behavior have a major influence on the interaction between the client and the component.*

A composition definition specifies a name that identifies the composition within the enclosing module scope, and which constitutes the name of a scope within which the contents of the composition are contained. The essential parts of a composition definition are the following:

- The name of the composition.
- The life cycle category of the component implementation, either **service**, **session**, **process**, or **entity**, as defined in Section 4.1.4, ÒComponent Categories,Ó on page 4-5.
- The home type being implemented (which implicitly identifies the component type being implemented).
- The name of the home executor to be generated.
- The name of the component executor skeleton to be generated.

A composition definition has the following essential form:

```
composition <category> <composition_name> {
    home executor <home_executor_name> {
        implements <home_type> ;
        manages <executor_name>;
    };
};
```

where <*composition_name*> is the name of the composition, <*category*> identifies the life cycle category supported by the composition, <*home_executor_name*> is the name assigned to the generated home executor skeleton, <*home_type*> is the name of a component home type imported from IDL, and <*executor_name*> is the name assigned to the generated component executor skeleton.

This is a schematic representation of the minimal form of a composition, which specifies no state management. The structure of the composition specified by this schematic is illustrated in Figure 3-1. Note that the component type itself is not explicitly specified. It is unambiguously implied by the specification of the home type, as is the relationship between the executor and the component (i.e., that the executor *implements* the component).

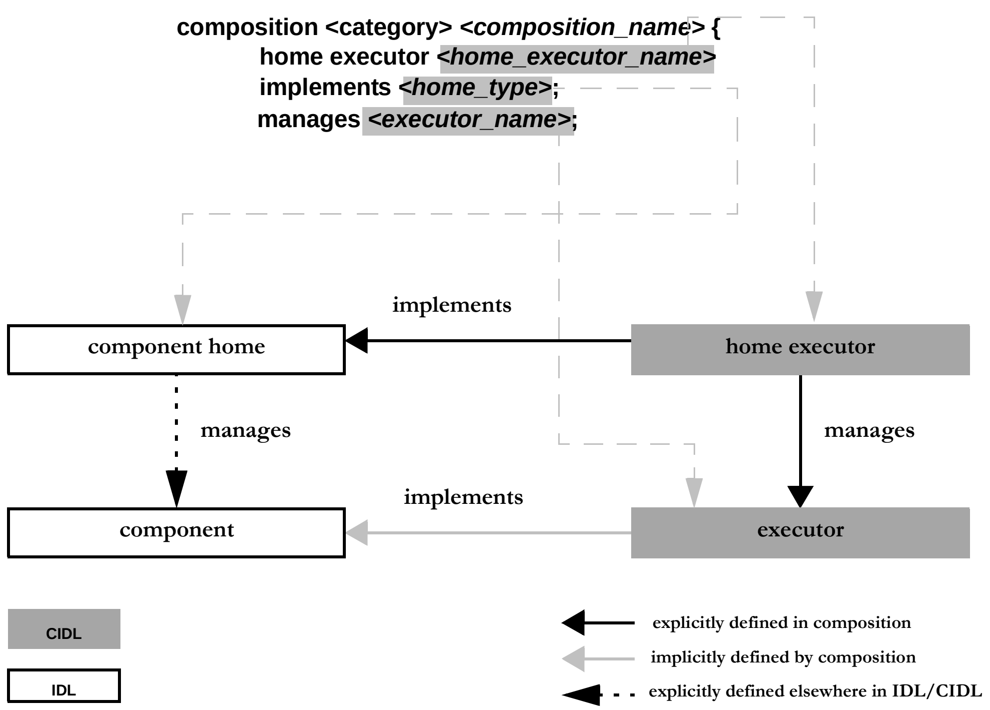


Figure 3-1 Minimal composition structure and relationships

General disclaimer and abdication of responsibility with regards to programming examples:

Before presenting programming examples, it should be noted that all examples are non-normative illustrations. In particular, the implementations provided in the examples of code that is to be generated by the CIF are merely schematic representations of the intended behaviors; they are by no means indicative of the actual content of a real implementation (e.g., they generally donÕt include exception handling, testing for validity, etc.).

Although the grammar for CIDL has not been presented yet, a simple example will help illustrate the concepts described in the previous sections. Assume the following IDL component and home definitions:

```
--------------------------------------------------------------------------------------
// Example 1
//
// USER-SPECIFIED IDL
//
module LooneyToons {
    interface Bird {
        void fly (in long how_long);
    };
    interface Cat {
        void eat (in Bird lunch);
    };
    component Toon {
```

```
            provides Bird tweety;
            provides Cat sylvester;
      };
      home ToonTown manages Toon {};
};
-----------------------------------------------------------------------------------
```

The following example shows a minimal CIDL definition that describes an implementation binding for those IDL definitions:

```
-----------------------------------------------------------------------------------
// Example 1
//
// USER-SPECIFIED CIDL
//
import ::LooneyToons;

module MerryMelodies {

      // this is the composition:

      composition session ToonImpl {
            home executor ToonTownImpl {
                  implements LooneyToons::ToonTown;
                  manages ToonSessionImpl;
            };
      };
};
-----------------------------------------------------------------------------------
```

In this example, ToonImpl *is the name of the composition. It defines the name of the generated home executor to be* ToonTownImpl, *which implemented the* ToonTown *home interface imported from IDL. The home executor definition also specified the name of the component executor,* ToonSessionImpl, *which is managed by the home executor. Note that the component type (*Toon*) is not explicitly namedÑit is implied by the specification of the home* ToonTown, *which is known to manage the component type Toon. Thus, the declaration Ò*manages ToonSessionImpl*Ó implicitly defines the component executor* ToonSessionImpl *to be the implementation of the component type* Toon.

This CIDL specification would cause the generation of the following artifacts:

¥ *The skeleton for the component executor* ToonSessionImpl

¥ *The complete implementation of the home executor* ToonTownImpl

We provide the following brief sketches of generated implementation skeletons in Java to help illustrate the programming model for component implementations.

Java <interface>Operations *interfaces for all of the IDL interfaces are generated, precisely as currently specified by the current Java IDL language mapping:*

```
-------------------------------------------------
// Example 1
//
// GENERATED FROM IDL SPECIFICATION:
//
package LooneyToons;

import org.omg.Components.*;

public interface BirdOperations {
   public void fly (long how_long);
}

public interface CatOperations {
   void eat(LooneyToons.Bird lunch);
}

public interface ToonOperations
extends CCMObjectOperations {
   LooneyToons.Bird provide_tweety();
   LooneyToons.Cat provide_sylvester();
}

public interface ToonTownExplicitOperations
extends CCMHomeOperations { }

public interface ToonTownImplicitOperations
extends KeylessCCMHomeOperations {
   Toon create();
}

public interface ToonTownOperations extends
ToonTownExplicitOperations,
ToonTownExplicitOperations {}
--------------------------------------------------
```

The ToonImpl executor skeleton class has the following form:

```
-------------------------------------------------
// Example 1
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;
import org.omg.Components.*;

abstract public class ToonSessionImpl
implements ToonOperations, SessionComponent,
ExecutorSegmentBase
{
   // Generated implementations of operations
   // inherited from SessionComponent and
   // ExecutorSegmentBase are omitted here.
   //
```

```
   protected ToonSessionImpl() {
      // generated implementation ...
   }

   // The following operations must be implemented
   // by the component developer:

   abstract public BirdOperations
      _get_facet_tweety();
   abstract public CatOperations
      _get_facet_sylvester();
}
--------------------------------------------------
```

The generated executor abstract base class ToonSessionImpl implements all of the operations inherited by ToonOperations, including operations on CCMObject and its base interfaces. It also implements all of the operations inherited through `SessionComponent`*, which are internal operations invoked by the container and the internals of the home implementation to manage executor instance lifecycle.*

A complete implementation of the home executor ToonTownImpl is generated from the CIDL specification:

```
--------------------------------------------------
// Example 1
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;
import org.omg.Components.*;

public class ToonTownImpl
implements LooneyToons,ToonTownOperations,
HomeExecutorBase, CCMHome
{
   // Implementations of operations inherited
   // from ExecutorBase and CCMHome
   // are omitted here.
   //
   // ToonHomeImpl also provides implementations
   // of operations inherited from the component
   // home interface ToonTown

   CCMObject create_component()
   {
      return create();
   }

   void remove_component(CCMObject comp)
   {
   }

   Toon create()
   {
```

```
    }
    // and so on...
}
------------------------------------------------
```

The user-provided executor implementation must supply the following:

- *Implementations of the operations* `_get_tweety` *and* `_get_sylvester`*, which must return implementations of the* `BirdOperations` *and* `CatOperations` *interfaces*
- *said implementations of the behaviors of the facets* tweety *and* sylvester*, respectively*

The following example shows one possible implementation strategy:

```
-------------------------------------------------
// Example 1
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonImpl extends ToonSessionImpl
implements BirdOperations, CatOperations {

    protected long timeFlown;
    protected Bird lastBirdEaten;

    public myToonImpl() {
        super();
        timeFlown = 0;
        lastBirdEaten = nil;
    }

    public void fly (long how_long) {
        timeFlown += how_long);
    }
    public void eat (Bird lunch) {
        lastBirdEaten = lunch;
    }
    public BirdOperations _get_facet_tweety() {
        return (BirdOperations) this;
    }
    public CatOperations _get_facet_sylvester() {
        return (CatOperations) this;
    }
}
--------------------------------------------------
```

This simple example implements all of the facets directly on the executor. This is not the only option; the programming objects that implement BirdOperations *and* CatOperations *could be constructed separately and managed by the executor class.*

The final bit of implementation that the component programmer must pro-

vide is an extension of the home executor that acts as a component executor factory, by implementing the create_executor_segment *method. This class must also provide an implementation of a static method called* create_home_executor *that returns a new instance of the home executor (as an* ExecutorSegmentBase*). This static method acts as an entry point for the entire composition.*

```
-----------------------------------------------
// Example 1
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonTownImpl extends ToonTownImpl
{
   protected myToonTownImpl() { super(); }

   ExecutorSegmentBase
   create_executor_segment (int segid) {
      return new myToonImpl();
   }

   public static ExecutorSegmentBase
   create_home_executor() {
      return new myToonTownImpl();
   }
}
--------------------------------------------------
```

Note that these last two classes constitute the entirety of the code that must be supplied by the programmer. The implementations of operations for navigation, executor activation, object reference creation and management, and other mechanical functions are either generated or supplied by the container.

3.2.7 Compositions with Managed Storage

A composition definition may also contain a variety of optional specifications, most of which are related to state management. These include the following elements:

Note Ð Issue 5499

¥ ~~One or more catalogs that provide the storage homes to the composition implementation. Each specified catalog is assigned an alias, or label, that identifies the catalog within the context of the composition.~~

¥ An abstract storage home type to which the component home is bound (this implicitly identifies the abstract storage type to which the component itself is bound).

¥ The life cycle category of the composition must be either **entity** or **process** to support managed storage.

When state management is added to a composition definition, the definition takes the following general form, expressed as a schematic:

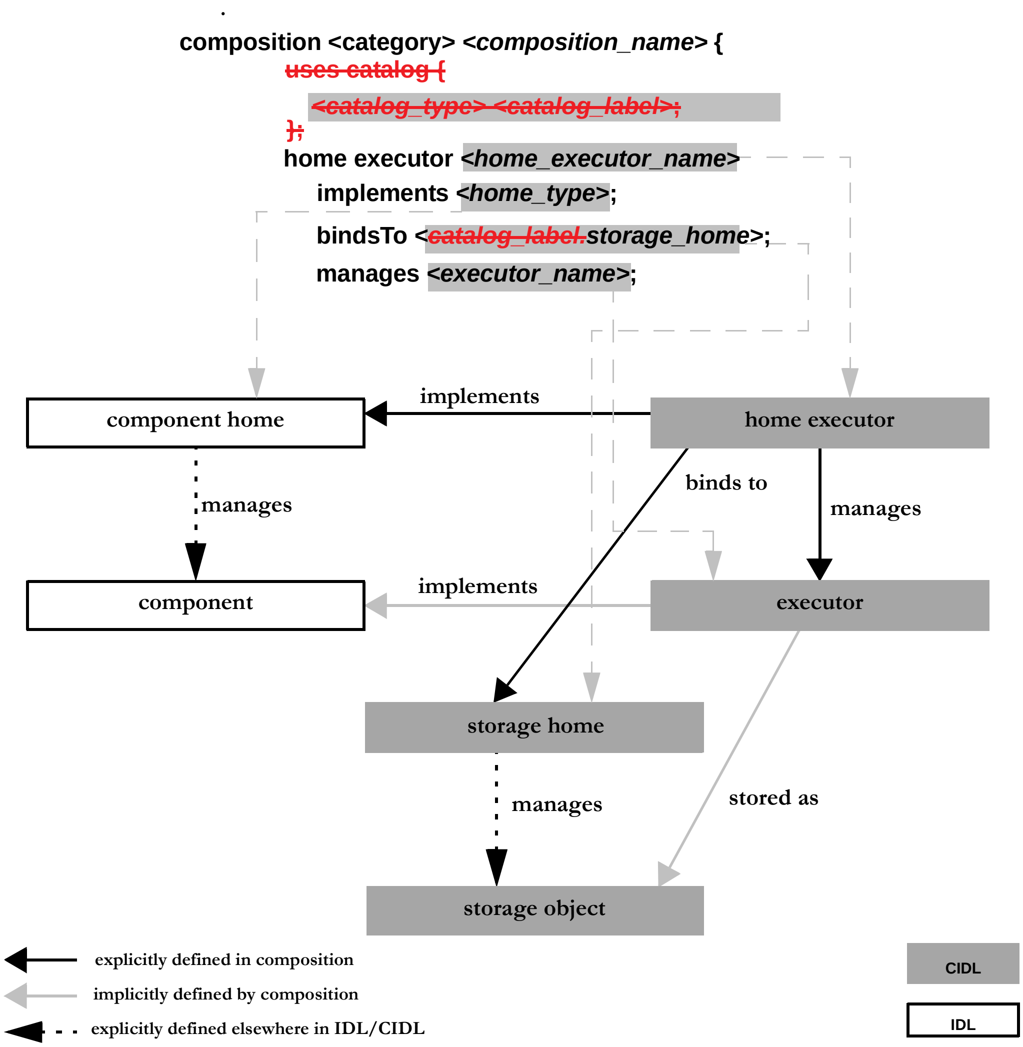
Note Ð Issue 5499

```
composition <category> <composition_name> {
    ~~uses catalog {~~
        ~~<catalog_type> <catalog_label>;~~
    ~~};~~
    home executor <home_executor_name> {
        implements <home_type> ;
        bindsTo <~~catalog_label.~~abstract_storage_home>;
        manages <executor_name>;
    };
};
```

where the additional element~~s are~~ is as follows: ~~<*catalog_type*> identifies the type of a catalog previously defined in PSDL, <*catalog_label*> is an alias by which the catalog can be identified in the composition definition, and~~ <~~*catalog_label.*~~*abstract_storage_home*> denotes a particular abstract storage home ~~provided by the catalog~~.

The structure of the resulting composition and the relationships between the elements is illustrated in Figure 3-2.

Note Ð Issue 5499



```
.
composition <category> <composition_name> {
        ~~uses catalog {~~
          ~~<catalog_type> <catalog_label>;~~
        ~~};~~
        home executor <home_executor_name>
            implements <home_type>;
            bindsTo <~~catalog_label.~~storage_home>;
            manages <executor_name>;
```

Figure 3-2 Structure of composition with managed storage

In many cases, it is expected that an abstract storage home will be intentionally designed to support a particular component home.

3.2.8 Relationship between Home Executor and Abstract Storage Home

When a composition specifies managed storage, the relationship between the home executor and the abstract storage home to which the home executor binds determines many of the characteristics of the implementation, including what implementation elements may be generated and how they will behave. This section provides an overview of the basic concepts involved in home implementations and their relationships to abstract storage homes.

In general, operations on a home interface provide life cycle management. As described in Section 1.7, ÒHomes,Ó on page 1-32, when a home definition does not specify a primary key, the resulting equivalent home interface has the following operations:

- A generic **create_component** operation inherited from **KeylessCCMHome,**
- a **remove_component** operation inherited from **CCMHome**, and
- an implicitly-defined type-specific parameter-less **create** operation.

When a home definition specifies a primary key, the resulting equivalent home interface has the following operations:

- A **remove_component** operation inherited from **CCMHome**,
- an implicitly-defined type-specific **create** operation with a primary key parameter,
- an implicitly-defined type-specific **remove** operation with a primary key parameter, and
- an implicitly-defined type-specific **find_by_primary_key** operation.

3.2.8.1 *Primary Key Binding*

A component home can define its primary key as a valuetype with a number of public data members, whereas abstract storage home defines keys as lists of attributes. A composition can only bind a component home with a primary key to an abstract storage home that defines a key on a state member whose type is this valuetype. If there is more than one key satisfying this condition, the first key is used.

For example:

```
valuetype SSN {
    public string social_security_number;
};

abstract storagetype Person {
    readonly state SSN social_security_number;
    state string name;
    state string address;
};

abstract storagehome PersonStore of Person {
    key social_security_number;
};
```

A home with primary key SSN can be bound to **PersonStore**. The key **social_security_number** is called the matching key.

3.2.8.2 Implicit delegation of home operations

When a composition specifies managed storage, finder operations can be implemented in terms of finder operations on the abstract storage home to which the home executor is bound.

Table 3-1 Delegation of finder operations to finder operations on the bound abstract storagehome

home operation	**abstract storagehome operation**
component find_by_primary_key (*key*)	*ref<X>* find_ref_by_*matching_key_name* (*matching_key*)

- The **find_by_primary_key** operation uses the **find_ref_by_*matching_key_name*** operation on the abstract storagehome. The returned storage reference is used to create an object reference for the component and returned to the invoking client.
- Destruction operations delegate to **destroy_object** operations on the reference.

The validity of these implementation semantics are predicated on the following assumptions:

- The initial state of the storage object created by the storage home constitutes a valid initial state for the component.
- All of the persistent state of the component is defined on (or reachable from) the storage object whose PID is associated with the component instance.
- The executor is monolithic, not segmented. Home operations can also be delegated to abstract storage homes when the executor is segmented, but the process is slightly more complex, and is discussed in full in Section 3.2.9.1, ÒSegmented executors,Ó on page 3-27.

If these assumptions do not hold (in particular, either of the first two), the component implementor can provide custom implementations of one or more home operations to accommodate the implementation requirements.

The following example extends the previous example to illustrate managed storage and storage home delegation. The example highlights differences from the previous, and does not repeat elements that are identical:

```
-------------------------------------------------------------------------------------
// Example 2
//
// USER-SPECIFIED IDL
//
module LooneyToons { // IDL

    ...  identical to previous example, except for the addition of the
         primary key:

    valuetype EpisodeName : Components::PrimaryKeyBase {
```

```
        public string name;
    };
    home ToonTown manages Toon primarykey EpisodeName {
    };
};
-----------------------------------------------------------------------------------
```

Note Ð Issue 5499

The CIDL now defines an abstract storage type~~s,~~ and an abstract storage home~~s, and a catalog~~. The composition binds :

```
-----------------------------------------------------------------------------------
// Example 2
//
// USER-SPECIFIED CIDL
//
import ::LooneyToons;

module MerryMelodies {

    abstract storagetype ToonState {
        state LooneyToons::EpisodeName episode_name;
        state string name;
        state unsigned long time_flown;
        state LooneyToons::Bird last_bird_eaten;
    };

    abstract storagehome ToonStateHome of ToonState
    {
        key episode_name;
        factory create(episode_name);
    };

    ~~catalog ToonCatalog {~~
        ~~provides ToonStateHome TSHome;~~
    ~~};~~

    // this is the composition:

    composition entity ToonImpl {
        ~~uses catalog { ToonCatalog store; };~~
        home executor ToonTownImpl {
            implements LooneyToons::ToonTown~~ {~~ ;
            bindsTo ~~store.TSHome~~ToonStateHome;
            manages ToonEntityImpl;
        };
    };
};
-----------------------------------------------------------------------------------
```

In this example, the composition binds the component home ToonTown *to the abstract storage home* ToonStateHome*, and thus, implicitly binds the component type* Toon *to the abstract storage type* ToonState*. Note that the primary key (if any) in the home must match a key in the abstract stor-*

age home. As will be seen later in the CIDL grammar specification, the keyword entity *in the implementation binding declaration specifies a particular lifecycle model for the resulting implementation.*

This CIDL specification would cause the generation of the following programming objects:

- *The skeleton for the component executor* ToonEntityImpl
- *The implementation of the home executor* ToonTownImpl
- *The incarnation interface for the abstract storage type* ToonState

Note Ð Issue 5499

- *The interface for the abstract storage home* ToonStateHome.
- ~~*The interface for the catalog ToonCatalog.*~~

Note that the complete implementation of the home executor may not be able to be generated in some cases, e.g., when no abstract storage type is declared or when user-defined operations with arbitrary signatures appear on the component home definition.

Note also that the implementations of the storage-related interfaces ToonState *and* ToonStateHome *are not necessarily provided by the same product that generates the component implementation skeletons. The CIF is specifically designed to decouple the executor implementation from the storage implementation, so that these capabilities may be provided by different products. A component-enabled ORB product is only required to generate the programming interfaces for the abstract storage type and homes through which the executor implementation will interact with one or more storage mechanisms. The implementations of these interfaces may be supplied separately, perhaps deferred until run-time.*

The interfaces generated from the IDL are identical, with the exception of the addition of the primary key:

```
--------------------------------------------------
// Example 2
//
// GENERATED FROM IDL SPECIFICATION:
//
package LooneyToons;

import org.omg.Components.*;

... same as previous except for the following:

public interface ToonTownImplicitOperations {
    Toon create(LooneyToons.EpisodeName key)
        throws DuplicateKey, InvalidKey;
    Toon find_by_primary_key
        (LooneyToons.EpisodeName key)
        throws UnknownKey, InvalidKey;
    void remove(LooneyToons.EpisodeName key)
        throws UnknownKey, InvalidKey;
```

```
    LooneyToons.EpisodeName
        get_primary_key(Toon comp);
}

public interface ToonTownOperations extends
ToonTownExplicitOperations,
ToonTownExplicitOperations {}
--------------------------------------------------
```

The abstract storage type ToonState results in the generation of the following incarnation interfaces:

```
--------------------------------------------------
// Example 2
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import org.omg.CosPersistentState.*;
import LooneyToons.*;

public interface ToonState extends StorageObject {
    public string name();
    public void name (String val);
    public long time_flown();
    public void time_flown (long val);
    public Bird last_bird_eaten();
    public void last_bird_eaten (Bird val);
}
```

The storage home ToonStateHome results in the generation of the following interface:

```
--------------------------------------------------
// Example 2
//
// GENERATED FROM CIDL SPECIFICATION:
//

// no explicit operations
public interface ToonStateHome
    extends StorageHomeBase {

    public ToonState
    find_by_episode_name (EpisodeName k);

    public ToonStateRef
    find_ref_by_episode_name (EpisodeName k);
}

--------------------------------------------------
```

The ToonImpl executor skeleton class has the following form:

```
--------------------------------------------------
// Example 2
```

```
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

abstract public class ToonImpl
implements LooneyToons.ToonOperations,
ExecutorSegmentBase, PersistentComponent
{
   // Generated implementations of operations
   // inherited from CCMObject and
   // ExecutorSegmentBase and PersistentComponent
   // are omitted here.
   //
   // ToonImpl also provides implementations of
   // operations inherited from ToonState, that
   // delegate to a separate incarnation object:

   protected ToonStateIncarnation _state;

   protected ToonImpl() { _state = null; }

   public void set_incarnation (ToonState state) {
      _state = state;
   }

   // The following operations must be implemented
   // by the component developer:

   abstract public BirdOperations
      _get_facet_tweety();
   abstract public CatOperations
      _get_facet_sylvester();
}
--------------------------------------------------
```

An implementation of the home executor ToonHomeImpl is generated from the CIDL specification:

```
--------------------------------------------------
// Example 2
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

public class ToonTownImpl
implements LooneyToons.ToonTownOperations,
PersistentComponent, ExecutorSegmentBase
{
   // Implementations of operations inherited
   // from PersistentComponent and
   // ExecutorSegmentBase
   // are omitted here.
```

```
    //
    // ToonHomeImpl also provides implementations
    // of operations inherited from the component
    // home interface ToonTown, that delegate
    // designated operations on the storage home
    //

    // values set during initialization
    // and activation:
    protected Entity2Context _origin;
    protected ToonStateHome _storageHome;
    ...

    Toon create(EpisodeName key)
    {
        // create a storage object with the key

        ToonStatenew_state=_storageHome.create(key);

        // REVISIT - Bernard Normier 7/27/1999
        // don’t know how to complete this method
    }

    Toon find(EpisodeName key)
    {
        ToonStateRef ref =
        _storageHome.find_ref_by_episode_name(key);
        // create reference from ref
        // and return , same as above...
    }

    // and so on...

}
------------------------------------------------
```

The user-provided executor uses the storage accessors and mutators on the incarnation:

```
------------------------------------------------
// Example 2
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonImpl extends ToonImpl
implements BirdOperations, CatOperations {

    public myToonImpl() { super(); }

    void fly (long how_long) {
        _state.timeFlown
            ( _state.timeFlown() + how_long);
    }
```

```
   void eat (Bird lunch) {
      _state.last_bird_eaten(lunch);
   }
   BirdOperations get_facet_tweety() {
      return (BirdOperations) this;
   }
   CatOperations get_facet_sylvester() {
      return (CatOperations) this;
   }
}
```

3.2.8.3 Explicit delegation of home operations

The previous section described the default home executor implementation generated by the CIF. Default delegation can only be implemented for home operations or the home base interfaces, and implicitly-defined home operations (i.e., *orthodox* home operations). The syntax for home definitions permits explicitly-defined factory operations, finder operations, and operations with arbitrary signatures to be declared on the home. The CIF makes no assumptions about the semantics of these operations (i.e., the *heterodox* operations), other than the assumptions that factory operations return references for newly-created components, and finder operations return references for existing components that were indirectly identified by the parameters of the finder operation. Implementations of these operations are not generated by default. CIDL does, however, allow the component implementor to specify explicitly how heterodox home operations are implemented. A CIDL home executor definition may optionally include the declarations illustrated in the following schematic CIDL example:

```
composition <category> <composition_name> {
    ...
    home executor <home_executor_name> {
        ... // assume storage management specified

        delegatesTo abstract storagehome (
            <home_op0> : <storage_home_op0>,
            <home_op1> : <storage_home_op1>, ...
        );
        delegatesTo executor(
            <home_op2> : <executor_op2>, ...
        );
        abstract(<home_op3>, <home_op4>, ...);
    };
};
```

Delegation to abstract storage home

The **delegatesTo abstract storagehome** declaration specifies a sequence of operation mappings, where each operation mapping specifies the name of an operation on the home, and the name of an operation on the storage home. The signatures of the operations must be compatible, as defined in Section 1.7.4, ÒHome inheritance,Ó on page 1-38. Based on this declaration, the CIF generates implementations of the home operations on the home executor that delegate to the specified operations on the abstract storage home.

Delegation to executor

The **delegatesTo executor** declaration specifies a sequence of operation mappings, similar to the **delegatesTo abstract storagehome** declaration. The name on the left hand side of the mapping (i.e., to the left of the colon, Ô:Õ) must denote an explicitly-declared factory operation on the home, or the identifier Ò**create**,Ó denoting the implicitly-declared factory operation. The right hand side of each mapping specifies the name of an abstract operation that will be generated on the component executor. The component implementor provides the implementation of the executor operation, and the CIF provides an implementation of the operation on the home executor that delegates to the executor.

The delegation of home operations to executors is problematic, since home operations (other than factories) have no target component. For this reason, only factory operations may be delegated to the component executor. The CIF implements this delegation by defining an additional facet on the component executor, called a *factory facet*. A factory facet is only exposed to the home executor; clients cannot navigate to the factory facet, and the factory facet is not exposed in component meta-data, or described in the **FacetDescription** values returned from **Navigation::get_all_facets**.

The implementation of the factory operation on the home executor that delegates to the component executor must first create an object reference that denotes the factory facet. The home operation then invokes the mapped factory operation on the object reference, causing the activation of the component and ensuring that the execution of the operation on the component occurs in a proper invocation context.

If the factory operation being delegated is any operation other than the orthodox **create** operation, and the home definition includes a primary key specification, the operation generated on the factory facet of the component executor returns a value of the specified primary key type. The delegating operation on the home executor associates the primary key value returned from the component executor with the storage object (i.e., the storage objectÕs PID) created to incarnate the component instance.

> *The use of PID values to create object references obviates the need to have two versions of a create method on the executor, as is the case in EJB with create and postCreate methods. An appropriate calling context can be created before the factory operation is invoked on the executor.*

These precise semantics of and requirements for factory operations delegated to the executor are described in detail in Section 1.7.3.1, ÒFactory operations,Ó on page 1-37.

Suppressing generated implementation

The **abstract** specification overrides the generation of implementations for orthodox home operations. The name of any explicitly-defined operation on the home may be specified in the operation list of the abstract declaration. The CIF will not implement the specified operations, instead leaving unimplemented abstract operation declarations (on whatever appropriate equivalent exists for the particular language mapping).

The following example extends the previous example to illustrate delegation of home operations to the abstract storage home and the executor. The example highlights differences from the previous, and does not repeat elements that are identical:

```
------------------------------------------------------------------------------------
// Example 3
//
// USER-SPECIFIED IDL
//
module LooneyToons { // IDL

    ...   identical to previous example, except for the home:

    home ToonTown manages Toon primarykey EpisodeName {
        factory createToon(
            in string name, in long num, in Bird bref);
        void arbitrary_operation();
    };
};
------------------------------------------------------------------------------------
```

Note Ð Issue 5499

The CIDL now defines abstract storage types~~,~~ and abstract storage homes~~, and a catalog~~. The composition binds:

```
------------------------------------------------------------------------------------
// Example 3
//
// USER-SPECIFIED CIDL
//
import ::LooneyToons;

module MerryMelodies {

    ... identical to the previous example, except for:

    abstract storagehome ToonStateHome of ToonState
    {
        key episode_name;
        factory create();
        void do_something();
    };

    composition entity ToonImpl {
        ~~uses catalog { ToonCatalog store; };~~
```

```
            home executor ToonTownImpl {
                implements LooneyToons::ToonTown;
                bindsTo ~~store.TSHome~~ToonStateHome;
                manages ToonEntityImpl;
                delegatesTo abstract storagehome
                    (arbitrary_operation : do_something);
                delegatesTo executor ( createToon : createToon );
            };
        };
};
------------------------------------------------------------------------------------
```

In this example, the arbitrary_operation on the home interface ToonTown is delegated to the storage home operation do_something. Note that the operations have identical signatures. The createToon factory operation is delegated to an operation of the same name on the executor. This delegation causes the implicit definition of a factory facet on the component with the following interface:

```
------------------------------------------------------------------------------------

interface ToonImplFactoryFacet {
    EpisodeName createToon(
            in string name, in long num, in Bird bref);
};
------------------------------------------------------------------------------------
```

This interface is not part of the public interface of the component; its use is restricted to the home executor. In fact, the IDL need not be generated. All of the code that uses the factory facet is either generated by the CIF, or derived from CIF-generated skeletons, so the CIF can simply generate language mappings for the interface without actually providing any IDL for it. Note also that only a subset of the normal language mapping artifacts are required, including (in the case of Java) the abstract Operations interface, the POA tie class to be used internally by the executor, and a local stub to allow the home executor to make a delegating invocation. There is no need to generate a remote stub, as the facet is never exposed outside of the container.

The abstract storage home ToonStateHome interface has the added do_something operation on the explicit interface:

```
------------------------------------------------
// Example 3
//
// GENERATED FROM CIDL SPECIFICATION:
//
public interface ToonStateHome
extends StorageHomeBase {
    public void do_something();
    // ...
}

--------------------------------------------------
```

The ToonImpl executor skeleton class supports an additional facet (the factory facet), which is returned by the _get_factory_facet operation:

```
--------------------------------------------------
// Example 3
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

abstract public class ToonImpl
implements LooneyToons.ToonOperations,
ExecutorSegmentBase, PersistentComponent {
   ... same as previous
   // The following operations must be implemented
   // by the component developer:

   abstract public ToonImplFactoryFacetOperations
      _get_factory_facet();
   abstract public BirdOperations
      _get_facet_tweety();
   abstract public CatOperations
      _get_facet_sylvester();
}
--------------------------------------------------
```

The CIF generates implementations of the delegated operations on the home executor:

```
--------------------------------------------------
// Example 3
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

public class ToonTownImpl
implements LooneyToons.ToonTownOperations,
CCMHome, ExecutorSegmentBase

   // values set during initialization
   // and activation:
   protected ToonStateHome _storageHome;
   protected Entity2Context _origin;

   ...

   Toon createToon(
      String name, long num, Bird bref)
   {
      ToonState new_state=
         _storageHome.create();
      // etc.
   }

   void arbitrary_operation() {
      _storageHome.do_something();
```

```
    }

    ...

}
------------------------------------------------
```

The user-provide executor must implement the factory facet and operation:

```
------------------------------------------------
// Example 3
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonImpl extends ToonImpl
implements BirdOperations, CatOperations,
ToonImplFactoryFacetOperations{
    ...
    ...

    EpisodeName
    createToon(String name, long num, Bird bref) {
        // presumably, the main reason for doing
        // this kind of delegation is to initialize
        // state in the context of the component:
        how_long(num);
        last_bird_eaten(bref);
        EpisodeNameDefaultFactory _keyFactory
            = new EpisodeNameDefaultFactory();
        return _keyFactory.create(name);
    }

    ToonImplFactoryFacetOperations
    _get_factory_facet() {
        return
            (ToonImplFactoryFacetOperations) this;
    }
    ...
}
```

3.2.9 Executor Definition

The home executor definition must include an executor definition. An executor definition specifies the following characteristics of the component executor:

- The name of the executor, which is used as the name of the generated executor skeleton.
- Optionally, one or more distinct segments, or physical partitions of the executor. Each segment encapsulates independent state and is capable of being independently activated. Each segment also provides at least one facet.

- Optionally, the generation of operation implementations that manage the state of stateful component features (i.e., receptacles, attributes, and event sources) as members of the component incarnation.
- A delegation declaration that describes a correspondence between stateful component features and members of the abstract storage type that incarnates the component. The CIF uses this declaration to generate implementations of the feature-specific operations (e.g., **connect_** and **disconnect_** operations for receptacles, accessors, and mutators for attributes) that store the state associated with each specified feature in the storage member indicated on the right hand side of the delegation.

3.2.9.1 Segmented executors

A component executor may be *monolithic* or *segmented*. A monolithic executor is, from the containerÕs perspective, a single artifact. A segmented executor is a set of physically distinct artifacts. Each segment may have a separate abstract state declaration. Each segment must provide at least one facet defined on the component definition. The life cycle category of the composition must be **entity** or **process** if the executor specifies segmentation.

The primary purpose for defining segmented executors is to allow requests on a subset of the componentÕs facets to be serviced without requiring the entire component to be activated. Segments are independently activated. When the container receives a request whose target is a facet of a segmented executor, the container activates only the segment that provides the required facet.

The following schematic CIDL illustrates the declaration of a segmented executor:

Note Ð Issue 5499

```
composition <category> <composition_name> {
    ...
    home executor <home_executor_name> {
        ... // assume storage management specified
        ...
        manages <executor_name> {
            segment <segment_name0> {
                storedOn <~~catalog_label.~~abstract_storage_home>;
                provides ( <facet_name0> , <facet_name1> , ... );
            };
            segment <segment_name1> { ... };
            ...
    };
};
```

The abstract storage home specified in the segmentÕs **storedOn** declaration implicitly specifies the abstract storage type that incarnates the segment. The home executor will use this abstract storage home to create and manage instances of the segment state (i.e., incarnations). If the component home specifies a primary key, then all of the abstract storage homes associated with executor segments must specify a matching key. The facets specified in the segmentÕs **provides** declaration are implemented on the segment.

A segmented executor has a distinguished segment associated with the component. The component segment is implicitly declared, and supplies all of the facets not provided by separate segments, as well as all other component features and supported interfaces.

Figure 3-3, and Figure 3-4, illustrate the structure of monolithic and segmented executors, and the relationships between facets, storage objects, and segments. These figures also illustrate the identity information that is embedded in component and facet object references. Component identity information is described in more detail in Section 1.1.4, ÒComponent Identity,Ó on page 1-4.

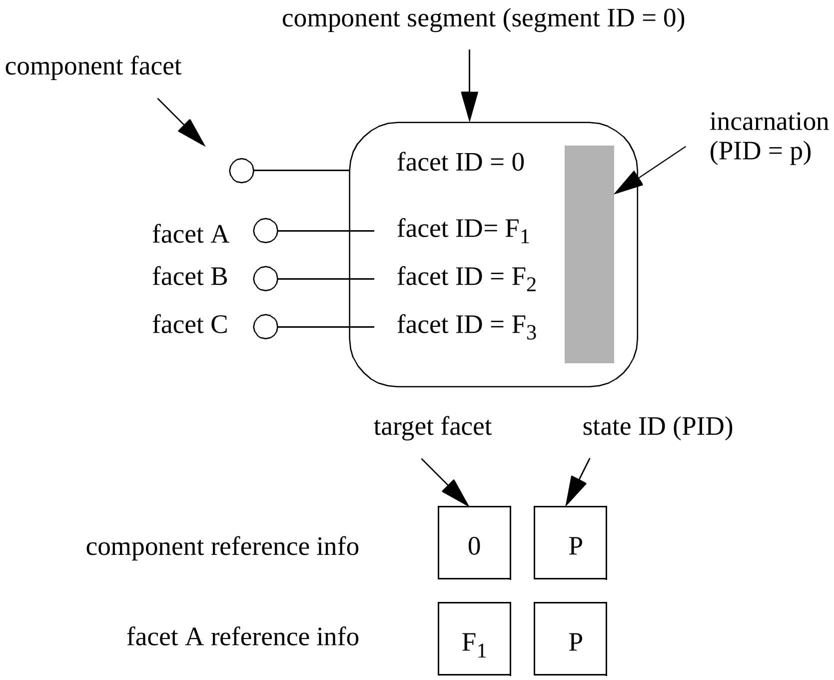


Figure 3-3 Monolithic executor and reference information structure

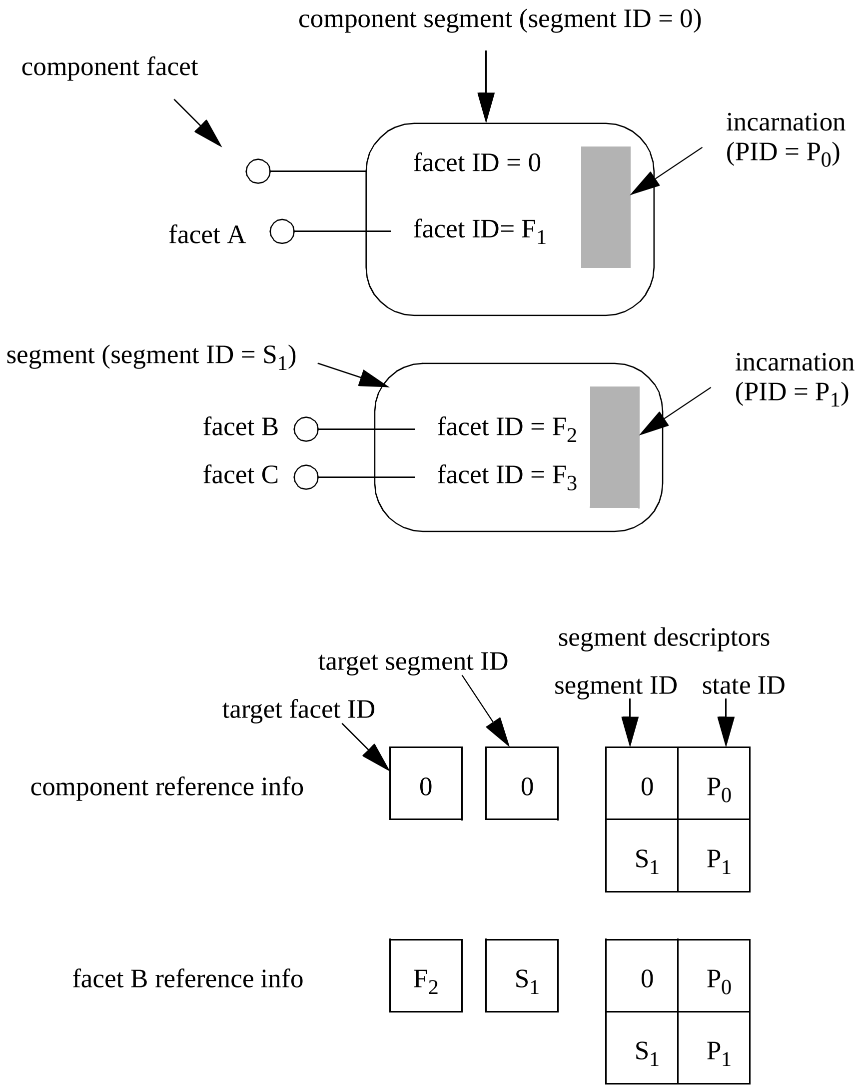


Figure 3-4 Segmented executor and reference information structure

The details of the structure and behavior of segments and requirements for their implementation are specified in Section 3.2.9.1, ÒSegmented executors,Ó on page 3-27.

> *The following example extends the previous example 2 to illustrate segmented executors. The example highlights differences from the previous, and does not repeat elements that are identical:*

```
--------------------------------------------------------------------------------------
//
// USER-SPECIFIED IDL
//
```

```
module LooneyToons { // IDL
    ...  identical to previous example 2
};
-------------------------------------------------------------------------------------
```

Note Ð Issue 5499

The CIDL now defines abstract storage types~~,~~ and abstract storage homes~~, and a catalog~~. The composition binds :

```
-------------------------------------------------------------------------------------
//
// USER-SPECIFIED CIDL
//
import ::LooneyToons;

module MerryMelodies {

    ... identical to example 2 except for new storage, storage home
        and executor definitions

    abstract storagetype ToonState {
        state LooneyToons::EpisodeName episode_name;
        state string name;
        state LooneyToons::Bird last_bird_eaten;
    };

    abstract storagehome ToonStateHome of ToonState {
        key episode_name;
    };  };

    abstract storagetype BirdSegState {
        state unsigned long time_flown;
    };

    abstract storagehome BirdSegStateHome of BirdSegState {
        key episode_name;
    };

    ~~catalog ToonCatalog {~~
        ~~provides ToonStateHome TSHome;~~
        ~~provides BirdSegStateHome BSSHome;~~
    ~~};~~

    composition entity ToonImpl {
        ~~uses Catalog { ToonCatalog store; };~~
        home executor ToonTownImpl {
            implements LooneyToons::ToonTown~~{~~ ;
            bindsTo ~~store.TSHome~~ToonStateHome;
            manages ToonEntityImpl {
                segment BirdSegment {
                    storedOn ~~ToonPS.BSSHome~~BirdSegState-
Home;
                    provides (tweety);
```

```
                };
            };
        };
    };
};
------------------------------------------------------------------------------------
```

The storage home BirdSegStateHome ~~BSSHome on the ToonCatalog catalog~~ is bound to the segment BirdSegment, which implicitly binds the segment executor for BirdSegment to the abstract storage type BirdSegState. This segment provides the facet tweety, leaving the remaining facet (sylvester) on the component segment.

The mappings of the CIDL abstract storage types~~,~~ and abstract storage homes~~, and the catalog~~ are not presented, as they are not affected by the segmentation.

The generated component executor base class ToonImpl is also not presented, as the changes are trivial. The facet accessor _get_facet_tweety is no longer present on the component executor. There are other internal changes that are not visible to the component implementor. The executor for the new BirdSegment has the following form:

```
------------------------------------------------
// Example 4
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

abstract public class BirdSegment
implements ExecutorSegmentBase,
PersistentComponent
{
    // Generated implementations of operations
    // inherited from CCMObject and
    // ExecutorSegmentBase and PersistentComponent
    // are omitted here.
    //

    protected BirdSegState _state;

    protected BirdSegment() { _state = null; }

    public void set_incarnation (
        BirdSegState state) {
        _state = state;
    }

    // The following operations must be implemented
    // by the component developer:

    abstract public BirdOperations
        _get_facet_tweety();
}
```

```
--------------------------------------------------
```

Note that the BirdSegment executor does not implement any IDL interface directly, as does the component segment. It is remotely accessible only through a provided facet.

A generated implementation of the home executor ToonHomeImpl is considerably different from the previous example 2. The create method must create references for all of the segments and construct a ComponentId with the proper information::

```
--------------------------------------------------
//
// GENERATED FROM CIDL SPECIFICATION:
//
package MerryMelodies;
import LooneyToons;

public class ToonTownImpl
implements LooneyToons.ToonTownOperations,
CCMHome, ExecutorSegmentBase
{
   // Implementations of operations inherited
   // from CCMHome and ExecutorSegmentBase
   // are omitted here.
   //
   // ToonHomeImpl also provides implementations
   // of operations inherited from the component
   // home interface ToonTown, that delegate
   // designated operations on the storage home
   //

   // values set during initialization
   // and activation:
      protected Entity2Context _origin;
   protected ToonStateHome _toonStorageHome;
   protected BirdSegStateHome _birdStorageHome;
   ...

   Toon create(EpisodeName key)
   {
      ToonState new_toon =
         _toonStorageHome.create(key);
      // etc.
   }
------------------------------------------------
```

There are now two segment executors to implement:

```
--------------------------------------------------
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonImpl extends ToonImpl
```

```
implements CatOperations {

   public myToonImpl() { super(); }

   void fly (long how_long) {
      _state.timeFlown
      ( _state.timeFlown() + how_long);
   }
   void eat (Bird lunch) {
      _state.last_bird_eaten(lunch);
   }
   BirdOperations get_facet_tweety() {
      return (BirdOperations) this;
   }
   CatOperations get_facet_sylvester() {
      return (CatOperations) this;
   }
}

public class myBirdSegImpl extends BirdSegment
implements BirdOperations {

   public myBirdSegImpl() { super(); }

   void fly (long how_long) {
      _state.timeFlown
      ( _state.timeFlown() + how_long);
   }

   BirdOperations get_facet_tweety() {
      return (BirdOperations) this;
   }
}
```

The programmer must also supply a different implementation of the create_executor_segment operation on the home executor, that uses the segment ID value to determine which executor to create.

```
// Example 4
//
// PROVIDED BY COMPONENT PROGRAMMER:
//
import LooneyToons.*;
import MerryMelodies.*;

public class myToonTownImpl extends ToonTownImpl
{
   protected myToonTownImpl() { super(); }

   ExecutorSegmentBase
   create_executor_segment (int segid) {

      // case discriminator values are constants
```

```
            // generated on the executor segment classes
            switch (segid) {
            case ToonImpl._segment_id_value :
               return new myToonImpl();
            case BirdSegment._segment_id_value :
               return new myBirdSegImpl();
            default
               ... raise an exception
            }
         }
         ...
      }
      -------------------------------------------------
```

3.2.9.2 Delegation of feature state

An executor may also optionally declare a correspondence between stateful component features (which include receptacles, attributes, and event sources) and members of the abstract storage type that incarnates the component (or the distinguished component segment, in the case of a segmented executor). The CIF uses this declaration to generate implementations of the feature-specific operations (e.g., connect_ and disconnect_ operations for receptacles, accessors, and mutators for attributes) that store the state associated with each specified feature in the storage member indicated on the right hand side of the delegation. The following schematic CIDL illustrates a feature delegation:

```
composition <category> <composition_name> {
   ...
      home executor <home_executor_name> {
      ... // assume storage management specified
      ...
      manages <executor_name> {
         delegatesTo abstract storagetype (
            <feature_name0> : <storage_member_name0> ,
            <feature_name1> : <storage_member_name1> , ...
         };
      };
   };
};
```

The type of the storage member must be compatible with the type associated with the feature, as defined in the *Component Model* chapter. In the case of attributes, the CIF-generated implementations of accessors and mutators retrieve and store the attribute value in the specified storage member. The executor programming model allows implementors to intercept invocations of the generated accessor and mutator invocations and replace or extend their behaviors. In the case of receptacles and event sources, the implementations of the **connect_<receptacle_name>, disconnect_<receptacle_name>, connect_<source_name>, disconnect_<source_name>, subscribe_<source_name>**, and **unsubscribe_<source_name>** operations store the connected object references in the specified members of the storage object that incarnates the component.

This mechanism is only particularly useful if the connected object references are persistent references, capable of causing server and object activation if necessary.

3.2.10 Proxy Homes

A composition definition may include a *proxy home* declaration. A proxy home implements the component home interface specified by the composition definition, but the implementation is not required to be collocated with the container where the components managed by the home are activated.

Proxy homes are, in essence, remote projections of the actual home implementation, which is always collocated with the executing component implementation. A proxy home may be able to implement some subset (or potentially, all) of the operations defined on the component home without contacting the actual home implementation. Operations that cannot be locally implemented by the proxy home are delegated to the actual home. The run-time implementation of the CIF (including the supporting infrastructure of the container and the home finder) is responsible for maintaining the associations between proxy homes and the actual home they represent. The container provides an interface for registering proxy homes, described in Section 4.4.1.3, ÒThe ProxyHomeRegistration Interface,Ó on page 4-35.

Proxy homes offer the capacity for considerably increased scalability over collocated homes, particularly when the home operations can be implemented locally by the proxy home implementation. The following schematic CIDL illustrates a proxy home definition:

Note Ð Issue 5499

```
composition <category> <composition_name> {
    ...
        home executor <home_executor_name> {
        implements <home_type> ;
        bindsTo <~~catalog_label.~~abstract_storage_home>;
        ...
    };
    proxy home <proxy_executor_name> {
        delegatesTo home ( <home_op0>, <home_op1>, ... );
        abstract (<home_op2>, <home_op3>, ...);
    };
};
```

The <***proxy_executor_name***> is used as the name of the generated skeleton artifact for the proxy home executor. The proxy home declaration implicitly acquires the characteristics of the actual home, as declared in the home executor definition (which must precede the proxy home definition in the composition scope). In particular, the proxy home implements the same home, and binds to the same abstract storage home.

The operation delegations specified in the actual home executor definition are also acquired by the proxy home, but certain delegations are transformed according to rules specified in Section 3.2.10.1, ÒProxy home delegation,Ó on page 3-36.

3.2.10.1 Proxy home delegation

For proxy homes in compositions that specify managed state, the CIF assumes that the proxy home has connectivity to the same persistent store as the actual home. Based on this assumption, the default implementations of orthodox operations on the proxy home executor are delegated directly to the storage home, precisely as they are in the actual home executor. In general, other operations are delegated to the actual home, by default, although the specific rules for determining the implementation of proxy home operations are somewhat more involved, and are described completely in Section 3.2.3, ÒImplementing a CORBA Component,Ó on page 3-2.

3.2.11 Component Object References

The CIF defines an information model for component object references. This information model is encapsulated within the object_key field of an IIOP profile, or an equivalent field in other profiles. The information model is an abstraction; no standard encoding within an **object_key** is specified. It is the responsibility of the container and the underlying ORB to encode this information for insertion into object references and to extract this information from the **object_key** in incoming requests, decode it, and use it to activate the appropriate component or segment and dispatch the request to the proper facet.

The **Entity2Context** interface, described in Section 4.4.3.7, ÒThe Entity2Context Interface,Ó on page 4-43 is used by the component implementation to provide this information to the container, with which the container creates the object references for the component and its facets. The **ComponentId** interface encapsulates the component reference information. Examples 2, 3, and 4 in the previous sections illustrate the use of the **Entity2Context** and **ComponentId** interfaces to create object references. Figure 3-3, and Figure 3-4, illustrate the structure of the information encapsulated in **ComponentId**, and its relationship to executor structure.

3.2.11.1 Facet identifiers

The CIF implementation allocates numeric identifiers to facets. The facet ID values are interpreted by generated code in the component implementation, so the assignment of values does not need to be uniformly specified; a given CIF implementationÕs choice of facet ID values does not affect portability or interoperability.

3.2.11.2 Segment identifiers

The CIF implementation must also allocate numeric identifiers to segments. Similar to facet IDs, segment IDs are also interpreted by the component implementation, so no uniform allocation mechanism is specified. The implementation of **create_executor_segment** (on the home executor implementation) provided by the component implementor must interpret segment ID values in order to create and return the appropriate segment executor. The generated implementations of segment executor skeletons define symbolic constants to assist the component implementor in this mapping.

3.2.11.3 State identifiers

State identifier is an abstraction that generalizes different representations of state identifiers, the primary of which is the **pid** of the CORBA persistent state service. The generic representation of a state identifier is **StateIdValue**, an abstract valuetype from which specific, concrete state identity types are derived. Implementations of the concrete sub-types are responsible for converting their representations to byte sequences and back again.

3.2.11.4 Monolithic reference information

Monolithic references contain a facet identifier and a single state identifier. The facet identifier denotes the target facet of the reference (or, of requests made on the reference). The state identifier is interpreted by the component implementation and used to retrieve the componentÕs state. In the case of automatically managed state, the CIF-generated implementation interprets the state identifier as a **pid**, using it to incarnate the componentÕs storage object.

> *Note that navigation from one facetÕs reference to another consists of merely replacing the target facet identifier with the facet identifier of the desired facet. This can be accomplished without activating the component.*

3.2.11.5 Segmented reference information

The reference information for segmented executors consists of the following:

- a target facet identifier
- a target segment identifier
- a sequence of segment descriptors, each of which contains:
  - the segment identifier of the segment being described
  - the state identifier for the segment

The target facet identifier denotes the target of requests made on the reference, and the target segment identifier denotes the segment on which that facet is implemented. The sequence of segment descriptors contains one element for each segment, including the component segment. This sequence is invariant for all references to a given component, over the lifetime of the component.

> *In the case of segmented executors, navigation is accomplished by replacing the facet and segment identifiers.*

3.2.11.6 Component identity

The state identifier of the component segment (or the single state identifier in the case of monolithic executors) is interpreted as the unique identity of the component, within the scope of the home to which it belongs. Equivalence of component identity is defined as equivalence of state identifier values of the component segment.

3.3 Language Mapping

3.3.1 Overview

This part describes the language mapping for CORBA Components and defines interfaces that are used to implement components and homes. The language mapping, like the mapping for the client side, is based on equivalent IDL. For components and homes, local interfaces are defined. The user then implements these local interfaces using existing language mapping rules.

There are two strategies for implementing a component, coined *monolithic* and *locator*. In the monolithic strategy, the user implements all attributes, supported interfaces, and event consumers in a single executor interface. In the locator strategy, the user implements a locator, and the container uses this locator to retrieve references to executors for each port of a component. The decision which strategy is being used is made by the home, which can return a reference to either the monolithic or to the locator.

It is expected that the monolithic strategy is more simple to use and that it is sufficient for most use cases, while the locator strategy gives the user even more control over the life cycle of each executor.

Interfaces are designated internal or callback. Callback interfaces are implemented by the user and called by the container, while internal interfaces are provided by the container.

Some callback interfaces may be optionally implemented by the user. In order to optionally implement an interface, the user must define an interface, in IDL, that inherits both the base interface and the optional interface. For example, to inherit the optional **SessionSynchronization** interface in the implementation of a **Bank** component, the user would declare a new local interface, as shown below.

```
local interface MyBank :
    Components::SessionSynchronization,
    CCM_Bank { };
```

Optional interfaces are used by services that require the componentÕs cooperation (and therefore callback hooks). To determine whether an implementation supports an optional interface, the container narrows the object reference to that interface.

Internal interfaces are used by the container and various services to provide runtime information to the component. The component accesses internal interfaces through the context reference that it acquires through the **set_session_context** operation.

Details about existing internal and callback interfaces can be found in the *Container Programming Model* chapter. Some of those interfaces are forward-referenced in this section.

3.3.2 Common Interfaces

EnterpriseComponent is an empty callback interface that serves as common base for all component implementations, whether monolithic or locator-based.

```
module Components {
    local interface EnterpriseComponent {};
};
```

Note Ð The **EnterpriseComponent** interface is also defined in the *Container Programming Model* chapter.

The **ExecutorLocator** interface is a callback interface that is used for the locator implementation strategy.

```
module Components {
    local interface ExecutorLocator : EnterpriseComponent {
        Object obtain_executor (in string name)
            raises (CCMException);
        void release_executor (in Object exc)
            raises (CCMException);
        void configuration_complete()
            raises (InvalidConfiguration);
    };
};
```

If a home, in creating a component, returns an **ExecutorLocator**, the container will invoke its **obtain_executor** operation prior to each invocation to retrieve the implementation for a port. The port name, given in the **name** parameter, is the same as used in the componentÕs interface description in IDL, or the componentÕs name for the ÒmainÓ executor. The **obtain_executor** operation returns a local object reference of the expected type, as detailed below. The CCMException exception may be raised in case of a system error that prohibits locating the requested executor.

The **release_executor** operation is called by the container once the current invocation on an executor that was obtained through the **obtain_executor** operation has finished. The locator can thus release any resources that were acquired as part of the **obtain_executor** operation.

The **configuration_complete** operation is called to propagate the **configuration_complete** operation on the **CCMObject** interface to the component implementation.

Implementations of the **ExecutorLocator** interface for a service or session component must implement the **Components::SessionComponent** interface. Implementations of the **ExecutorLocator** interface for a process or entity component must implement the **Components::EntityComponent** interface.

Note Ð Object is used as the return type of the **obtain_executor** operation, because there is yet no IDL type for the common base of all local objects. Since local objects inherit from **Object**, this is not a problem.

The **HomeExecutorBase** interface is a common base for all home implementations.

```
module Components {
    local interface HomeExecutorBase {};
};
```

3.3.3 Mapping Rules

This section defines equivalent interfaces that are generated for each interface, eventtype, component, and home.

3.3.3.1 Interfaces

For each non-abstract and non-local interface, a local *facet executor* interface is generated. This facet executor interface has the same name as the original interface with a Ò**CCM_**Ó prefix, and inherits the original interface. So for an interface of name ***<interface name>***, the facet executor interface has the following form:

```
local interface CCM_<interface name> : <interface name> { };
```

If a component provides an interface as a facet, the user implements the facet executor interface rather than the original interface in order to achieve a local implementation.

Note Ð A container implementation may choose to limit generation of facet executor interfaces to only those interfaces that are actually used as a facet.

3.3.3.2 Eventtypes

For each eventtype, a local *consumer executor* interface is generated. For an eventtype ***<eventtype name>***, a local interface with the same name, but with a Ò**CCM_**Ó prefix and a postfix of Ò**Consumer**Ó is generated. This interface has a single **push** operation with no result, and the eventtype as a single **in** parameter:

```
local interface CCM_<eventtype name>Consumer
{
    void push (in <eventtype name> ev);
};
```

3.3.3.3 Components

A component maps to three local interfaces; two of them are callback interfaces, and one is an internal interface. The *monolithic executor* callback interface is for use in monolithic implementations, the *main executor* callback interface is for use in locator-based implementations. Both callback interfaces inherit the componentÕs base and supported interfaces. They also both expose the componentÕs attributes.

In addition, the monolithic executor callback interface also contains operations for acquiring references to facets, and for consuming events - in the locator approach, these jobs are mediated by the locator.

An internal *context* interface is defined for each component. It is implemented by the container and handed to the component as session or entity context. The context interface contains component-specific runtime information (e.g., for pushing events into event source ports).

Component Main Executor Interface

The *main executor* callback interface as used by the locator approach is defined by the following rules:

1. For each component ***<component name>***, a local *main executor* interface with the same name as the component, but with a prefix of Ò**CCM_**Ó and a postfix of Ò**_Executor**Ó is defined.
2. The main executor interface contains all attributes declared by the component.
3. If the component has a base component with a name of ***<base name>***, the main executor interface inherits **CCM_*<base name>*_Executor**. If the component does not have a base, the main executor interface inherits **Components::EnterpriseComponent**.
4. If the component has supported interfaces, they are inherited by the main executor interface.

If the container desires to acquire a reference to the main executor, it calls the **obtain_executor** operation of the **ExecutorLocator** with the name parameter set to ***<component name>***.

Component Monolithic Executor Interface

The *monolithic executor* callback interface is defined by the following rules:

1. For each component ***<component name>***, a local *monolithic executor* interface with the same name as the component and a prefix of Ò**CCM_**Ó is defined.
2. The monolithic executor interface contains all attributes declared by the component.
3. If the component has a base component with a name of ***<base name>***, the monolithic executor interface inherits **CCM_*<base name>***. If the component does not have a base, the monolithic executor interface inherits **Components::EnterpriseComponent**.

4. If the component has supported interfaces, they are inherited by the monolithic interface.
5. Additional operations are added to the monolithic interface for facets and event sinks.
6. Above rules can be satisfied by inheriting the main executor interface and adding operations for facets and event sinks. This is an optional design choice by the container implementation.

In a service and session component, the user may optionally inherit the **Components::SessionComponent** interface in the implementation of a monolithic executor in order to be notified by the container of activation and passivation. In a process or entity component, the user may optionally inherit the **Components::EntityComponent** interface in the implementation of a monolithic executor.

Component Context Interface

The *context* internal interface is defined by the following rules:

1. For each component ***<component name>***, a local *context* interface with the same name as the component, but with a prefix of Ò**CCM_**Ó and a postfix of Ò**_Context**Ó is defined.
2. If the component has a base component with a name of ***<base name>***, the context interface inherits **CCM_*<base name>*_Context**. If the component does not have a base, the context interface inherits **Components::CCMContext**.
3. Additional operations are added to the context interface for receptacles and event sources.

The container will implement an interface that inherits both the above context interface and either **Components::SessionContext** or **Components::EntityContext**, depending on the type of the component. The component implementation can narrow the **Components::SessionContext** or **Components::EntityContext** reference that it receives to the above component-specific context interface.

3.3.3.4 Example

For the following component declaration in IDL,

```
interface Hello {
    void sayHello ();
};

component HelloWorld supports Hello {
    attribute string message;
};
```

the following local interfaces are generated:

```
local interface CCM_Hello : Hello
{
}

local interface CCM_HelloWorld_Executor :
    Components::EnterpriseComponent, Hello
{
    attribute string message;
};

local interface CCM_HelloWorld :
    Components::EnterpriseComponent, Hello
{
    attribute string message;
};

local interface CCM_HelloWorld_Context :
    Components::CCMContext
{
};
```

Read on for further contents of these interfaces.

3.3.3.5 Ports

This section defines equivalent operations that are added to either of the three interfaces for each port definition.

Facets

For each facet, an equivalent operation is defined in the monolithic executor interface. For a facet of name ***<name>*** and type ***<type>***, an operation with the same name as the facet but with a Ò**get_**Ó prefix is generated. This operation has an empty parameter list and a reference of the interfaceÕs facet executor type as return value:

CCM_*<type>* get_*<name>* ();

Users may optionally implement facet interfaces directly in the monolithic executor implementation by declaring a new local interface that inherits both the monolithic executor interface and the facet executor, and by then returning a reference to itself in the implementation of the above operation. Example:

```
// IDL
component MyComponent {
    provides MyInterface MyFacet;
};

// User IDL
local interface MyComponentImpl :
    CCM_MyComponent, CCM_MyInterface
{};

// C++
CCM_MyInterface_ptr
MyComponent_Impl::get_MyFacet ()
{
    return CCM_MyInterface::_duplicate (this);
}
```

If the locator strategy is used, the container calls the **obtain_executor** operation on the **ExecutorLocator** with the **name** parameter set to ***<name>*** in order to acquire a reference to the facet executor that matches this facet port.

Receptacles

For each receptacle, an equivalent operation is defined in the context interface. The signature of this operation depends on whether the receptacle is simplex or multiplex.

For a simplex receptacle of name ***<name>*** and type ***<type>***, an operation of the same name as the receptacle but with a Ò**get_connection_**Ó prefix is generated. The operation has an empty parameter list, and an object reference of the interfaceÕs type as return value:

```
<type> get_connection_<name> ();
```

If there is no connection, this operation returns a nil reference.

For a multiplex receptacle of name ***<name>*** and type ***<type>***, an operation of the same name as the receptacle but with a Ò**get_connections_**Ó prefix is generated. The operation has an empty parameter list and a sequence of type ***<name>*Connections** as return value (this type is defined by the client-side equivalent IDL):

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

Publisher and Emitter

For each publisher and emitter port, an equivalent operation is defined in the context interface. For a publisher or emitter port of name ***<name>*** and type ***<type>***, an operation of the same name as the port but with a Ò**push_**Ó prefix is generated. This operation has no return value and a single **in** parameter containing the event.

```
void push_<name> (in <type> ev);
```

The component may call this operation in order to push an event to the consumer (for emitter ports) or to all subscribers (for publisher ports). The container is responsible for delivering the event.

Consumer

For each consumer port, an equivalent operation is defined in the monolithic executor interface. For a consumer port of name ***<name>*** and type ***<type>***, an operation of the same name as the port but with a Ò**push_**Ó prefix is generated. This operation has no return value and a single **in** parameter containing the event.

void push_<*name*> (in <*type*> ev);

For component implementations that use the monolithic strategy, the container invokes this operation whenever a client sends an event to this sink.

For component implementations that use the locator strategy, the container calls the **obtain_executor** operation on the **ExecutorLocator** with the name parameter set to ***<name>*** in order to acquire a reference to an implementation of the eventtypeÕs consumer executor interface.

3.3.3.6 Home

For each home, three callback interfaces are generated, similar in structure to the interfaces defined on the client side. The three interfaces are named the *Implicit*, *Explicit*, and *Main* home executor.

Home Explicit Executor Interface

The *home explicit executor* callback interface is defined by the following rules:

1. For each home ***<home name>***, a local *explicit executor* interface with the same name as the home, but with a prefix of Ò**CCM_**Ó and a postfix of Ò**Explicit**Ó is defined.
2. The explicit executor interface contains all attributes and operations declared by the home.
3. If the home has a base with a name of ***<base name>***, the explicit executor interface inherits **CCM_<*base name*>Explicit**. If the home does not have a base, the explicit executor interface inherits **Components::HomeExecutorBase**.
4. If the home has supported interfaces, they are inherited by the explicit executor interface.
5. Additional operations are added to the explicit executor interface for factories and finders, see below.

Home Implicit Executor Interface

The contents of the *home implicit executor* callback interface depend on whether the home is keyless or keyed.

Implicit Executor Interface for Keyless Homes

For a *keyless* home ***<home name>***, a local implicit executor interface with the same name as the home, but with a prefix of Ò**CCM_**Ó and a postfix of Ò**Implicit**Ó is defined. This interface contains a single **create** operation with the following signature:

```
local interface CCM_<home name>Implicit {
    Components::EnterpriseComponent create ()
        raises (Components::CCMException);
};
```

The container calls the implicit **create** operation in order to create a new component instance. The operation can return either a reference to a monolithic executor or to an **ExecutorLocator**. In the former case, the container assumes that the monolithic strategy is used, otherwise it will use the locator strategy. The implementation may raise the CCMException exception in order to indicate a system-level error.

Note Ð Issue 5340

Implicit Executor Interface for Explicitly or Implicitly Keyed Homes

For a keyed home ***<home name>*** with a key of ***<key type>*** or a keyless home ***<home name>*** that derives from a keyed home with a key of ***<key type>***, a local implicit executor interface with the same name as the home, but with a prefix of Ò**CCM_**Ó and a postfix of Ò**Implicit**Ó is defined. This interface contains the following operations:

```
local interface CCM_<home name>Implicit {
    Components::EnterpriseComponent
        create (in <key type> key)
        raises (Components::CCMException);
    Components::EnterpriseComponent
        find_by_primary_key (in <key type> key)
        raises (Components::CCMException);
    void remove (in <key_type> key)
        raises (Components::CCMException);
};
```

The container calls the **create** operation in order to create a new component associated with the specified primary key value. The operation can return either a reference to a monolithic executor or to an **ExecutorLocator**. In the former case, the container assumes that the monolithic strategy is used, otherwise it will use the locator strategy. The operation may raise the CCMException exception to indicate a system-level error.

The container calls the **find_by_primary_key** operation in order to find an existing component associated with the specified primary key value. The operation shall return the same reference to a monolithic executor or to an **ExecutorLocator** as it was previously returned from a **create** operation. The operation may raise the CCMException exception to indicate a system-level error.

The container calls the **remove** operation in order to remove the component identified by the specified primary key value. The operation may raise the CCMException exception to indicate a system-level error.

Home Main Executor Interface

For each home ***<home name>***, a local *main executor* interface with the same name as the home and a prefix of Ò**CCM_**Ó is defined. The main executor interface inherits both the implicit and explicit executor interfaces, as shown below.

```
local interface CCM_<home name> :
    CCM_<home name>Explicit,
    CCM_<home name>Implicit
{
};
```

The main executor interface does not have any other contents.

In the implementation of a home main executor for a service and session component, the user may optionally inherit the **Components::SessionComponent** interface in order to be notified by the container of activation and passivation. In the implementation of a home main executor for a process or entity component, the user may optionally inherit the **Components::EntityComponent** interface.

Note Ð This structure allows implementation inheritance for the explicit interface without name clashes in the implicit interface.

Factories

For each factory in the home, an operation is defined in the explicit home executor interface. This operation has the same parameter list as the factory and the return type **EnterpriseComponent**. As with the homeÕs **create** operation, factories can return either a reference to a monolithic executor or to an **ExecutorLocator**.

Factories are assumed to return a new component instance.

Finders

For each finder in the home, an operation is defined in the explicit home executor interface. This operation has the same parameter list as the finder and the return type **EnterpriseComponent**. As with the homeÕs **create** operation, finders can return either a reference to a monolithic executor or to an **ExecutorLocator**.

Finders may return existing or new component instances. If a finder decides to return an existing component instance, it shall return the same reference to a monolithic executor or to an **ExecutorLocator** as it was previously returned from a factory or from the **create** operation.

Entry Points

Some programming languages require the existence of user-provided entry points, or *Home Factories*. These entry points are not part of the language mapping; they are dealt with in the *Packaging and Deployment* chapter.

Home Factories, if required by a language mapping, shall return a reference to an instance of the homeÕs main executor interface.

Example

The following example shows a **Bank** home that manages an **Account** component.

```
home Bank manages Account {
    factory open (in string name);
    void close (in string name);
};
```

In this example, the following equivalent interfaces would be generated.

```
local interface CCM_BankExplicit :
    Components::HomeExecutorBase
{
    Components::EnterpriseComponent open (in string name);
    void close (in string name);
};

local interface CCM_BankImplicit :
{
    Components::EnterpriseComponent create ()
        raises (Components::CCMException);
};

local interface CCM_Bank :
    CCM_BankExplicit,
    CCM_BankImplicit
{
};
```

The user would then implement the **CCM_Bank** interface and eventually provide an entry point that creates a **CCM_Bank** instance.

The Container Programming Model 4

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter describes the CORBA component container programming model.

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒIntroductionÓ	4-2
ÒThe Server Programming EnvironmentÓ	4-5
ÒServer Programming Interfaces - Basic ComponentsÓ	4-21
ÒServer Programming Interfaces - Extended ComponentsÓ	4-33
ÒThe Client Programming ModelÓ	4-45

The container is the serverÕs 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 Òqualities of serviceÓ characteristics and have no effect on the set of interfaces the component implementor can rely on. This chapter is organized as follows:

¥ Section 4.1 - ÒIntroductionÓ 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 4.2 - ÒThe Server Programming EnvironmentÓ 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 4.3 - ÒServer Programming Interfaces - Basic ComponentsÓ 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 the *Component Model* chapter, which can be used by servers as well as clients, they define the server programmerÕs API.

¥ Section 4.5 - ÒThe Client Programming ModelÓ describes the client view of a CORBA component.

The **client programming model** has been described previously (see the *Component Model* chapter). This section describes the specific use of CORBA required by a client, which is **NOT** itself a CORBA component, to use a CORBA component written to the server programming model described in Section 4.3, ÒServer Programming Interfaces - Basic Components,Ó on page 4-21.

4.1 Introduction

The container programming model is made up of several elements:

¥ The **external API types** define the interfaces available to a component client.

¥ The **container API type** defines the API framework used by the component developer.

¥ The CORBA usage model defines the interactions between the container and the rest of CORBA (including the POA, the ORB, and the CORBA services).

¥ The component category is the combination of the container API type (i.e., the server view) and the external API types (i.e., the client view).

The overall architecture is depicted in Figure 4-1.

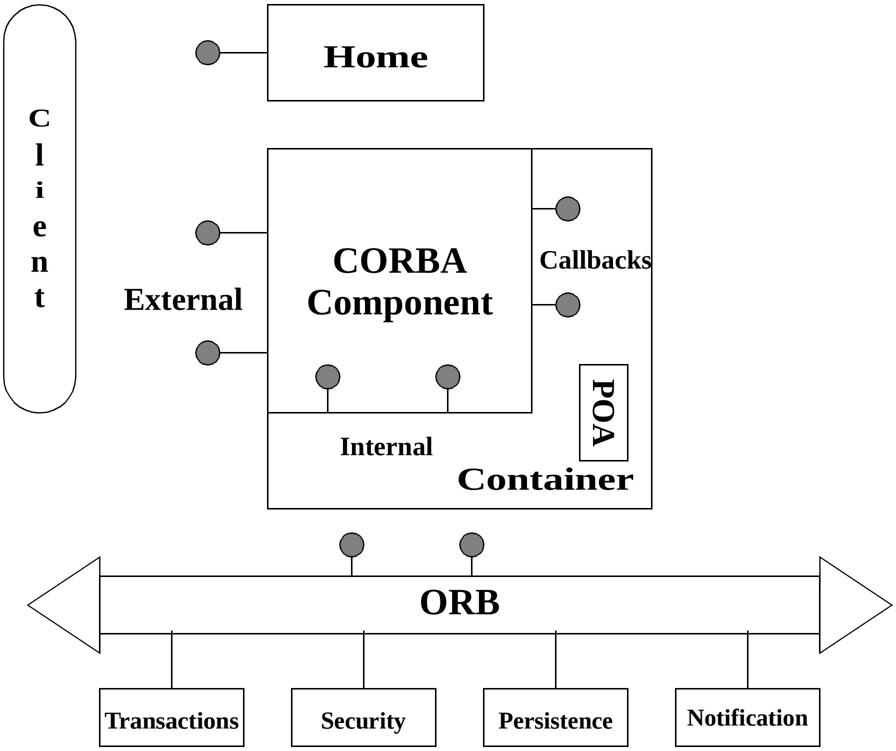


Figure 4-1 The Architecture of the Container Programming Model

The external API 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 API type** is a framework made up of internal interfaces and callback interfaces used by the component developer. These are defined using the new **local interface** declaration in IDL for specifying locality-constrained interfaces. The container API type is selected using CIDL, which describes component implementations.

> *The EJB session bean and entity bean can be viewed as two examples of container API type since they offer different sets of framework APIs to the EJB programmer. However, each of them also implies a client view (i.e., the external API types). EJB does not define a term for the two framework API sets it supports.*

The CORBA usage model is controlled by policies that specify distinct interaction patterns with the POA and a set of CORBA services. These are defined by CIDL, augmented using XML, and used by the container factory to create a POA when the container is created.

The component category is a specific combination of external API types and container API type used to implement an application with the CORBA component technology.

4.1.1 External API Types

The external API types of a component are the contract between the component developer and the component client. We distinguish between two forms of external API types: the **home** interface and the **application** interfaces.

> *These are analogous to the* **EJBHome** *and* **EJBObject** *interfaces of Enterprise JavaBeans.*

Home interfaces support operations that allow the client to obtain references to one of the application interfaces the component implements. From the clientÕs perspective, two design patterns are supported - factories for creating new objects and finders for existing objects. These patterns are distinguished by the presence of a **primarykey** parameter in the home IDL declaration.

¥ A home interface with a **primarykey** declaration supports finders and its client is a **keyfull** client.

¥ A home interface without a **primarykey** declaration does not support finders and its client is a **keyless** client. All home types support factory operations.

4.1.2 Container API Type

The **container API type** defines an API framework; that is, the contract between a specific component and its container. This specification defines two base types that define the common APIs and a set of derived types that provide additional function. The **session** container API type defines a framework for components using transient object references. The **entity** container API type defines a framework for components using persistent object references.

4.1.3 CORBA Usage Model

A CORBA usage model specifies the required interaction pattern between the container, the POA, and the CORBA services. We define three **CORBA usage models** as part of this specification. Since all support the same set of CORBA services, they are distinguished only by their interaction with the POA.

¥ **stateless** - which uses transient object references in conjunction with a POA servant that 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.*

4.1.4 Component Categories

The component categories are defined as the valid combinations of external API types, container API type, and CORBA usage model. Table 4-1 summarizes the categories and identifies their EJB equivalent.

Table 4-1 Definition of the Component Categories

CORBA Usage Model	**Container API Type**	**Primary Key**	**Component Categories**	**EJB Bean Type**
stateless	session	No	Service	-
conversational	session	No	Session	Session
durable	entity	No	Process	-
durable	entity	Yes	Entity	Entity

4.2 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 that 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 4.3, ÒServer Programming Interfaces - Basic Components,Ó on page 4-21.

4.2.1 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 componentÕs 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, and
- 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 interface** types, which provide container functions to the CORBA component.

  > *These are similar to the* **EJBContext** *interface in Enterprise JavaBeans.*

- **Callback interfaces** - These are also **local interface** 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 JavaBeans.*

This architecture is depicted in Figure 4-1 on page 4-3.

We define a small set of **container API types** to support a broad spectrum of component behavior with their associated **internal** and **callback** interfaces as part of this specification. These **container API types** are defined using local interfaces.

Additional component behavior is controlled by policies specified in the deployment descriptor. This specification defines policies that support POA interactions (CORBA usage model), servant lifetime management, transactions, security, events, and persistence. See the *Packaging and Deployment* chapter, specifically Section 6.3, ÒSoftware Package Descriptor,Ó on page 6-2, for details of how container policies are specified.

CORBA containers are designed to be used as Enterprise JavaBeans containers. This allows a CORBA infrastructure to be the foundation of EJB, enabling a more robust implementation of the EJB specification. To support enterprise Beans natively within a CORBA container, the container must support the API frameworks defined by the EJB specification. This architecture is defined in the *Integrating with Enterprise JavaBeans* chapter of this specification.

4.2.2 CORBA Usage Model

The *CORBA Component Specification* defines a set of **CORBA usage models** that create either **TRANSIENT** or **PERSISTENT** object references and use either a 1:1 or 1:N mapping of **Servant** to **ObjectId**. These CORBA usage models are summarized in Table 4-2. A given component implementation shall support one and only one CORBA usage model.

Table 4-2 CORBA Usage Model Definitions

CORBA Usage Model	Object Reference	Servant:OID Mapping
stateless	**TRANSIENT**	1:N
conversational	**TRANSIENT**	1:1
durable	**PERSISTENT**	1:1
(Invalid)	**PERSISTENT**	1:N

A CORBA usage model is specified using CIDL and is used to either create or select a component container at deployment time.

4.2.2.1 *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 **self-managed** or **container-managed** persistence. **PERSISTENT** objects can be used with the CORBA persistent state service or any user-defined persistence mechanism. When the CORBA persistent state service is used, servant management is aligned with the **PersistentId** defined by the CORBA persistent state service 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).

Home references are exported for client use by registering them with a **HomeFinder** which the client subsequently interrogates or by binding them to the CORBA naming service in the form of externally visible names.

> *EJB clients find references to EJBHome using JNDI, the Java API for CosNaming. Placing home references is CosNaming supports both the CORBA component client and the EJB client programming models.*

4.2.2.2 *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** CORBA usage model, 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 4.2.5, ÒServant Lifetime Management,Ó on page 4-8) and therefore is stateful.

4.2.2.3 *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 CIDL.

> *A threading policy of* **serialize** *is required to support an enterprise Bean since they are defined to be single-threaded.*

4.2.3 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; for example, 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.

4.2.4 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 componentÕs 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 componentÕs descriptor, and invoke callback interfaces on the component as necessary.

4.2.5 Servant Lifetime Management

Servants are programming language objects that 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, the server programmer makes several design choices:

- The container API type must be chosen.
- The CORBA usage model 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 that 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.

Table 4-3 shows the relationship between the CORBA usage model, the container API type, and the servant lifetime policies.

Table 4-3 Servant Lifetime Policies by Container API Type

CORBA Usage Model	Container API Type	Valid Servant Lifetime Policies
stateless	session	**method**
conversational	session	**method**, **transaction**, **component**, **container**
durable	entity	**method**, **transaction**, **component**, **container**

Servant lifetime policies may be defined for each segment within a component.

4.2.6 *Transactions*

CORBA components may support either **self-managed transactions** (SMT) or **container-managed transactions** (CMT). A component using self-managed transactions will not have transaction policies defined with its deployment descriptor and is responsible for transaction demarcation using either the containerÕs **UserTransaction** interface or the CORBA transaction service. A component using container-managed transactions defines transaction policies in its associated descriptor. The selection of container-managed transactions vs. self-managed transactions is a component-level specification.

When container-managed transactions are selected, additional transaction policies are defined in the componentÕs deployment descriptor. The container uses these descriptions to make the proper calls to the CORBA transaction service. The transaction policy defined in the componentÕs deployment descriptor is applied by the container prior to invoking the operation. Differing transaction policy declarations can be made for operations on any of the componentÕs ports as well as for the componentÕs home interface.

Table 4-4 summarizes the effects of the various transaction policy declarations and the presence or absence of a client transaction on the transaction that is used to invoke the requested operation on the component.

Table 4-4 Effects of Transaction Policy Declaration

Transaction Attribute	**Client Transaction**	**ComponentÕs Transaction**
NOT_SUPPORTED	-	-
	T1	-
REQUIRED	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1
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 outside the scope of a transaction.

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 componentÕs transaction attempts to commit (**CosTransactions::Current::commit**), and the clientÕs 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 shall be raised.

never

This component requires that the client not be in a current transaction to execute successfully. If no current transaction exists, the operation is invoked. If a current transaction exists, the INVALID_TRANSACTION exception shall be raised.

4.2.7 Security

Security policy is applied consistently to all categories of components. The container relies on CORBA security to consume the security policy declarations from the deployment descriptor and to check the active credentials for invoking operations. The security policy remains in effect until changed by a subsequent invocation on a different component having a different policy.

Access permissions are defined by the deployment descriptor associated with the component. The granularity of permissions must be aligned by the deployer with a set of rights recognized by the installed CORBA security mechanism since it will be used to check permissions at operation invocation time. Access permissions can be defined for any of the componentÕs ports as well as the componentÕs home interface.

Note Ð The security model used by EJB and being adopted by CORBA components requires the secure transportation of security credentials between systems. Today that is only possible if SECIOP is used as the CORBA transport.

4.2.8 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 sending and removed from the event data structure when receiving.
- Components support two forms of event generation using the push model:
  - A component may be an exclusive supplier of a given type of event.
  - A component may supply events to a shared channel that other CORBA notification users are also utilizing.
- A CORBA component consumes both forms of events using the push model.
- Events have transaction and security policies associated with the componentÕs event ports as defined 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.

Because events can be emitted and consumed by clients as well as component implementations, 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.

4.2.8.1 Transaction Policies for Events

Transaction policies are defined for component event ports, which include both events being generated and events being consumed. The possible values are as follows:

normal

A **normal** event policy indicates the event should be generated 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 generated 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** event policy indicates the event should be generated 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.

Transaction policy declarations can be defined in the deployment descriptor for each event port defined by the component.

4.2.8.2 Security Policies for Events

CORBA components permits access control policies based on roles to be associated with the generation and consumption of events. This is accomplished by associating ACLs with the component ports used to emit/publish and consume events and using CORBA security to restrict access. These policies provide access control based on role for both event generation and consumption.

4.2.9 Persistence

The **entity** container API type supports the use of a persistence mechanism for making component state durable; for example, storing it in a persistent store like a database. The **entity** container API type defines two forms of persistence support:

- **container-managed persistence (CMP)** - the component developer simply defines the state that is to be made persistent 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 state declaration language of the CORBA persistent state service and connecting that state declaration to a component segment using CIDL.

- **self-managed persistence (SMP)** - the component developer assumes the responsibility for saving and restoring state when requested to do so by the container.

Self-managed persistence is selected via CIDL declaration and triggered by the container invoking the callback interfaces (which the component must implement) defined later in this chapter (Section 4.3, ÒServer Programming Interfaces - Basic Components,Ó on page 4-21).

Table 4-5 summarizes the choices and their required responsibilities.

Table 4-5 Persistence Support for Entity Container API 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
Self-managed	CORBA	Component	Generated Code	Component implements
Self-managed	User	Component	Component implements	Component implements

Container-managed vs. self-managed persistence is selected via the deployment descriptor for each segment of the component.

4.2.9.1 *Container-managed Persistence*

Container-managed persistence may be accomplished using the CORBA persistent state service or any user-defined persistence mechanism. When the CORBA persistent state service is used, the container manages all interactions with the persistence provider and the component developer need not use the persistence interfaces offered by the container. With container-managed persistence using the CORBA persistent state service, 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.

Container-managed persistence is selected using CIDL and tailored using XML at deployment time to specify connections to specific persistence providers and persistent stores.

4.2.9.2 *Self-managed Persistence*

Self-managed persistence is also supported by the **entity** container API type. Like container-managed persistence, the component developer has two choices: to use the CORBA persistent state service 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 supports access to a component persistence abstraction provided by the CORBA persistent state service, which hides many of the details of the underlying persistence mechanism from the component developer.

Self-managed persistence is selected using CIDL and tailored using XML at deployment time to specify connections to specific persistence providers and persistent stores.

4.2.10 Application Operation Invocation

The application operations of a component can be specified on both the componentÕs supported interfaces and the provided interfaces. These operations are normal CORBA object invocations.

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.

4.2.11 Component Implementations

A component implementation consists of one or more **executors**. Each **executor** describes the implementation characteristics of a particular component segment. The session container API type consists of a single **executor** with a single segment that is activated in response to an operation request on any component facet. The entity container API type can be made up of multiple segments, each of which is associated with a different abstract state declaration. Each segment is independently activated when an operation request on a facet associated with that segment is received.

4.2.12 Component Levels

The CORBA component specification defines two levels of component function that can be used by component developers and supported by CORBA container providers:

¥ **basic** - The basic CORBA component supports a single interface (or multiple interfaces related by inheritance) and does not define any ports (provided interfaces or event source/sinks). The implementation of a basic component may use transaction, security, and simple persistence (i.e., a single segment) and relies on its container to manage the construction of CORBA object references.

> *The basic component is functionally equivalent to the EJB 1.1 Component Architecture.*

- **extended** - The extended component is a basic component with multiple ports (supported interfaces, provided interfaces and/or event source/sinks). The implementation of the extended component may use all basic function, advanced persistence (multiple segments) plus the event model and participates in the construction of component object references.

The component interfaces defined in this specification have been structured into functional modules corresponding to the two levels of components defined above.

- Basic container APIs are defined in Section 4.3, ÒServer Programming Interfaces - Basic Components,Ó on page 4-21.
- Extended container APIs are defined in Section 4.4, ÒServer Programming Interfaces - Extended Components,Ó on page 4-33.

Partitioning the component function into two discrete packages permits the EJB 1.1 APIs to be used to implement basic CORBA components in Java. It also supports the construction of CORBA components in any supported CORBA language that can be accessed by EJB clients. This is described further in the ÒIntegrating with Enterprise JavaBeans" chapter.

4.2.13 Component Categories

As indicated in Section 4.1.4, ÒComponent Categories,Ó on page 4-5, this specification defines four component categories whose behavior is specified by the two **container API types**. Additionally we reserve a component category to describe the empty container (i.e., a container API type that does not use one of the API frameworks defined in this specification). The four component categories are described briefly in the following sections. The component categories are independent of the component levels defined in Section 4.2.12, ÒComponent Levels,Ó on page 4-15.

4.2.13.1 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 that have no duration 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 provides a simple way of wrapping existing procedural applications.

A service component is equivalent to a stateless EJB session bean.

Table 4-6 summarizes the characteristics of a service component as seen by the server programmer.

Table 4-6 Service Component Design Characteristics

Design Characteristic	**Property**
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus **SessionContext** (basic) **Session2Context** (extended)
Callback Interfaces	**SessionComponent**
CORBA Usage Model	stateless
External API Types	keyless
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 managed 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.

The service component can use either container-managed or self-managed transactions.

4.2.13.2 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 4.2.5, ÒServant Lifetime Management,Ó on page 4-8. 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. A session component is equivalent to the stateful session bean found in EJB.

Table 4-7 summarizes the characteristics of a session component as seen by the server programmer.

Table 4-7 Session Component Design Characteristics

Design Characteristic	**Property**
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus **SessionContext** (basic) **Session2Context** (extended)
Callback Interfaces	**SessionComponent** plus (optionally) **SessionSynchronization**
CORBA usage model	conversational
Client Design Pattern	Factory
External API Types	keyless
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 managed 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, however extended components may choose to participate in creating references if desired, and the client sees only the factory design pattern.

The session component shall use either container-managed or self-managed transactions.

4.2.13.3 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 can be 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).

Table 4-8 summarizes the characteristics of process component as seen by the server programmer.

Table 4-8 Process Component Design Characteristics

Design Characteristic	**Property**
External Interfaces	As defined in component IDL
Internal Interfaces	Base set plus **EntityContext** (basic) **Entity2Context** (extended)
Callback Interfaces	**EntityComponent**
CORBA usage model	durable
Client Design Pattern	Factory
External API Types	keyless
Persistence	Self-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 managed storage

A process component may have transactional behavior. The container will interact with the CORBA transaction service to participate in the commit process. The process component shall use container-managed transactions. This is identical to the EJB restriction for Entity Beans.

The process component can use **container-managed** or **self-managed** persistence using either the CORBA persistent state service or a user-defined persistence mechanism. The implications of the various choices are described in Section 4.2.9, ÒPersistence,Ó on page 4-13. The entity container uses callback interfaces, which enable the process componentÕs implementation to retrieve and save state data at activation and passivation respectively.

4.2.13.4 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 may be used to implement the entity bean in EJB.

Table 4-9 summarizes the characteristics of **entity** component as seen by the server programmer:

Table 4-9 Entity Component Design Characteristics

Design Characteristic	**Property**
External Interfaces	As defined in the component IDL
Internal Interfaces	Base set plus **EntityContext** (basic) **Entity2Context** (extended)
Callback Interfaces	**EntityComponent**
CORBA usage model	durable
Client Design Pattern	Factory or Finder
External API Types	keyfull
Persistence	Self-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 managed storage

The entity component shall use container-managed transactions. The container shall interact with the CORBA transaction service to participate in the commit process. This is identical to the EJB restriction for Entity Beans.

The entity component can use **container-managed** or **self-managed** persistence using either the CORBA persistent state service or a user-defined persistence mechanism. The implications of the various choices are described in Section 4.2.9, ÒPersistence,Ó on page 4-13. The entity container uses callback interfaces that enable the entity componentÕs implementation to retrieve and save state data at activation and passivation, respectively.

4.3 Server Programming Interfaces - Basic Components

This section defines the local interfaces used and provided by the component developer for basic components. These interfaces are then grouped as follows:

- Interfaces common to both container API types.
- Interfaces supported by the session container API type only.
- Interfaces supported by the entity container API type only.

Unless otherwise indicated, all of these interfaces are defined within the **Components** module.

4.3.1 Component Interfaces

All components deal with three sets of interfaces:

- **Internal** interfaces that are used by the component developer and provided by the container to assist in the implementation of the componentÕs behavior.
- **External** interfaces that are used by the client and implemented by the component developer.
- **Callback** interfaces that 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 API 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.

- **CCMContext** - serves as a bootstrap and provides accessors to the other internal interfaces including access to the runtime services implemented by the container.

  Each container API type has its own specialization of **CCMContext**, which we refer to as a context.
- **UserTransaction** - wraps the demarcation subset of the CORBA transaction service required by the application developer.
- **EnterpriseComponent** - the base class that all **callback** interfaces derive from.

  All components implement a callback interface that is determined by the component category. It serves the same role as EnterpriseBean in EJB.

When a component instance is instantiated in a container, it is passed a reference to its context, a local interface used to invoke services. For basic components, these services include transactions and security. The component uses this reference to invoke operations required by the implementation at runtime beyond what is specified in its deployment descriptor.

4.3.2 Interfaces Common to both Container API Types

This section describes the interfaces and operations provided by both **container API types** to support all categories of CORBA components.

4.3.2.1 The CCMContext Interface

The **CCMContext** is an **internal** interface that provides a component instance with access to the common container-provided runtime services applicable to both **container API types**. It serves as a ÒbootstrapÓ to the various services the container provides for the component.

The **CCMContext** 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.

```
typedef SecurityLevel2::Credentials Principal; exception IllegalState { };

local interface CCMContext {
    Principal get_caller_principal();
    CCMHome get_CCM_home();
    boolean get_rollback_only() raises (IllegalState);
    Transaction::UserTransaction get_user_transaction()
                                    raises (IllegalState);
    boolean is_caller_in_role (in string role);
    void set_rollback_only() raises (IllegalState);
};
```

get_caller_principal

The **get_caller_principal** operation obtains the CORBA security credentials in effect for the caller. Security on the server is primarily controlled by the security policy in the deployment descriptor for this component. The component may use this operation to determine the credentials associated with the current client invocation.

get_CCM_home

The **get_CCM_home** operation is used to obtain a reference to the home interface. The home is the interface that supports factory and finder operations for the component and is defined by the **home** declaration in component IDL.

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 IllegalState exception shall be 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.

get_user_transaction

The **get_user_transaction** operation is used to access the **Transaction::UserTransaction** interface. The **UserTransaction** interface is used to implement self-managed transactions. The IllegalState exception shall be raised if this component is using container-managed transactions.

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.

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 IllegalState exception shall be 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.

4.3.2.2 *The Home Interface*

A home is an **external** interface that supports factory and finder operations for the component. These operations are generated from the **home** IDL declaration (see Section 1.7, ÒHomes,Ó on page 1-32). The context supports an operation (**get_CCM_home**) to obtain a reference to the componentÕs home interface.

4.3.2.3 *The UserTransaction Interface*

A CORBA component may use either container-managed or self-managed transactions, depending on the component category. With container-managed transactions, the component implementation relies on the transaction policy declarations packaged with the deployment descriptor and contains no transaction APIs in its implementation code.

> *This is identical to container-managed transactions in EJB or the default processing of an MTS component.*

A component specifying self-managed transactions 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 shall 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 service 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 JavaBeans.*

The **UserTransaction** is an **internal** interface implemented by the container and is defined within its own module, **Transaction**, within the **Components** module **(Components::Transaction**). Because the **UserTransaction** is a wrapper over **CosTransactions::Current**, it is thread specific. The **UserTransaction** exposes a simple demarcation subset of the CORBA transaction service to the component. The context supports an operation (**get_user_transaction**) to obtain a reference to the **UserTransaction** interface. The **UserTransaction** interface is defined by the following IDL.

```
typedef sequence<octet> TranToken;
exception NoTransaction { };
exception NotSupported { };
exception SystemError { };
exception RollbackError { };
exception HeuristicMixed { };
exception HeuristicRollback { };
exception Security { };
exception InvalidToken { };

enum Status {
        ACTIVE,
        MARKED_ROLLBACK,
        PREPARED,
        COMMITTED,
        ROLLED_BACK,
```

```
        NO_TRANSACTION,
        PREPARING,
        COMMITTING,
        ROLLING_BACK
        };


local interface UserTransaction {
    void begin () raises (NotSupported, SystemError);
    void commit () raises (RollbackError , NoTransaction,
                            HeuristicMixed, HeuristicRollback,
                            Security, SystemError);
    void rollback () raises (NoTransaction, Security, SystemError);
    void set_rollback_only () raises (NoTransaction, SystemError);
    Status get_status() raises (SystemError);
    void set_timeout (in long to) raises (SystemError);
    TranToken suspend () raises (NoTransaction, SystemError);
    void resume (in TranToken txtoken)
                            raises (InvalidToken, SystemError);
};
```

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** with **report_heuristics** set to **TRUE** 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 NotSupported exception is returned if it is received from the CORBA transaction service. Since nested transactions are not supported by CORBA component containers, this indicates an attempt to start a new transaction when an existing transaction is active. All other exceptions are converted to the SystemError exception.

commit

The **commit** operation is used by a component to terminate an existing transaction normally. 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. If no transaction is active, the NoTransaction exception shall be raised. If the TRANSACTION_ROLLEDBACK system exception is returned, it is converted to the RollbackError exception. The **CosTransaction::HeuristicMixed** and **CosTransaction::HeuristicRollback** exceptions are reported as the **HeuristicMixed** and **HeuristicRollback** exceptions respectively. The NO_PERMISSION system exception is converted to the Security exception. All other exceptions are converted to the SystemError exception.

rollback

The **rollback** operation is used by a component to terminate an existing transaction abnormally. 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. If no transaction is active, the NoTransaction exception shall be raised. The NO_PERMISSION system exception is converted to the Security exception. All other exceptions are converted to the SystemError exception.

set_rollback_only

The **set_rollback_only** operation is used by a component to mark an existing transaction for abnormal termination. 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. If no transaction is active, the NoTransaction exception shall be raised. All other exceptions shall be converted to the SystemError exception.

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. All exceptions shall be converted to the SystemError exception.

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. All exceptions are converted to the SystemError exception.

suspend

The **suspend** operation is used by a component to disconnect an existing transaction from the current thread. The **suspend** operation returns a **TranToken**, 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.

If no transaction is active, the NoTransaction exception shall be raised. All other exceptions are converted to the SystemError exception.

resume

The **resume** operation is used by a component to reconnect a transaction previously suspended to the current thread. The **TranToken** identifies the suspended transaction that is to be resumed. If the transaction identified by **TranToken** has not been suspended, the InvalidToken exception shall be 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 single suspended transaction is the only transaction that may be resumed. All other exceptions are converted to the SystemError exception.

> *The* **UserTransaction** *interface is equivalent to the* **UserTransaction** *interface (***javax.transaction.UserTransaction***) in EJB with the addition of the* **suspend** *and* **resume** *operations.*

4.3.2.4 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 that defines no operations.

```
local interface EnterpriseComponent { };
```

4.3.3 Interfaces Supported by the Session Container API Type

This section describes the interfaces supported by the session container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces, which must be implemented by components deployed in this container API type.

4.3.3.1 The SessionContext Interface

The **SessionContext** is an **internal** interface that provides a component instance with access to the container-provided runtime services. It serves as a ÒbootstrapÓ to the various services the container provides for the component. The **SessionContext** enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };

local interface SessionContext : CCMContext {
        Object get_CCM_object() raises (IllegalState);
};
```

get_CCM_object

The **get_CCM_object** operation is used to get the reference used to invoke the component. For basic components, this will always be the component reference. For extended components, this will be a specific facet reference. If this operation is issued outside of the scope of a **callback** operation, the IllegalState exception is returned.

4.3.3.2 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.

```
enum CCMExceptionReason {
    SYSTEM_ERROR,
    CREATE_ERROR,
    REMOVE_ERROR,
    DUPLICATE_KEY,
    FIND_ERROR,
    OBJECT_NOT_FOUND,
    NO_SUCH_ENTITY};

exception CCMException {CCMExceptionReason reason;};

local interface SessionComponent : EnterpriseComponent {
    void set_session_context ( in SessionContext ctx)
                        raises (CCMException);
    void ccm_activate() raises (CCMException);
    void ccm_passivate() raises (CCMException);
    void ccm_remove () raises (CCMException);
};
```

set_session_context

The **set_session_context** operation is used to set the **SessionContext** 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.The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_activate

The **ccm_activate** 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.The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_passivate

The **ccm_passivate** 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. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_remove

The **ccm_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. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

4.3.3.3 The SessionSynchronization Interface

The **SessionSynchronization** interface is a **callback** interface that may optionally be implemented by the session component. It permits the component to be notified of transaction boundaries by its container.

```
exception CCMException {CCMExceptionReason reason;};

local interface SessionSynchronization {
    void after_begin () raises (CCMException);
    void before_completion () raises (CCMException);
    void after_completion (
    in boolean committed) raises (CCMException);
};
```

after_begin

The **after_begin** operation is called by the container to notify a session component that a new transaction has started, and that the subsequent operations will be invoked in the context of the transaction.The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

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. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

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 component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

4.3.4 Interfaces Supported by the Entity Container API Type

This section describes the interfaces supported by the entity container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces that must be implemented by components deployed in this container API type.

4.3.4.1 The EntityContext Interface

The **EntityContext** is an **internal** interface that provides a component instance with access to the container-provided runtime services. It serves as a ÒbootstrapÓ to the various services the container provides for the component.

The **EntityContext** enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };

local interface EntityContext : CCMContext {
    Object get_CCM_object () raises (IllegalState);
    PrimaryKeyBase get_primary_key () raises (IllegalState);
};
```

get_CCM_object

The **get_CCM_object** operation is used to obtain the reference used to invoke the component. For basic components, this will always be the component reference. For extended components, this will be a specific facet reference. If this operation is issued outside of the scope of a **callback** operation, the IllegalState exception is returned.

get_primary_key

The **get_primary_key** operation is used by an **entity** component to access the primary key value declared for this componentÕs home. This operation is equivalent to issuing the same operation on the componentÕs home interface. If this operation is issued outside of the scope of a **callback** operation, the IllegalState exception is returned.

4.3.4.2 The EntityComponent Interface

The **EntityComponent** is a **callback** interface implemented by both process and entity components. It contains operations to manage the persistent state of the component.

Note Ð As currently defined, any operation request will cause the container to activate the component segment, if required. Since the component reference is well-structured, we could consider the possibility of trapping navigation operations prior to activation and executing them without actually activating the component (or we could leave that to clever implementations).

```
exception CCMException {CCMExceptionReason reason;};

local interface EntityComponent : EnterpriseComponent {
    void set_entity_context (in EntityContext ctx)
            raises (CCMException);
    void unset_entity_context ()raises (CCMException);
    void ccm_activate () raises (CCMException);
    void ccm_load ()raises (CCMException);
    void ccm_store ()raises (CCMException);
    void ccm_passivate ()raises (CCMException);
    void ccm_remove ()raises (CCMException);
    };
```

set_entity_context

The **set_entity_context** operation is used to set the **EntityContext** 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. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

unset_entity_context

The **unset_entity_context** operation is used to remove the **EntityContext** 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. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_activate

The **ccm_activate** operation is called by the container to notify the component that it has been made active. For most CORBA component implementations, no action is required. The component instance should perform any initialization (other than establishing its state) required prior to operation invocation. This operation is called within an unspecified transaction context. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_load

The **ccm_load** operation is called by the container to instruct the component to synchronize its state by loading it from its underlying persistent store. When container-managed persistence is implemented using the CORBA persistent state service, this operation can be implemented in generated code. If self-managed persistence is being used, the component is responsible for locating its state in a persistent store. This operation executes within the scope of the current transaction. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_store

The **ccm_store** operation is called by the container to instruct the component to synchronize its state by saving it in its underlying persistent store. When container-managed persistence is implemented using the CORBA persistent state service, this operation can be implemented in generated code. If self-managed persistence is being used, the component is responsible for saving its state in the persistent store. This operation executes within the scope of the current transaction. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_passivate

The **ccm_passivate** operation is called by the container to notify the component that it has been made inactive. For most CORBA component implementations, no action is required. The component instance should perform any termination processing (other than saving its state) required prior to being passivated. This operation is called within an unspecified transaction context. The component may raise the CCMException with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_remove

The **ccm_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 component raises the CCMException with the **REMOVE_ERROR** minor code if it does not allow the destruction of the component. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

> *The* **EntityComponent** *interface is equivalent to the* **EntityBean** *interface in Enterprise JavaBeans. Container-managed persistence with the CORBA persistent state service supports automatic code generation for* **ccm_load** *and* **ccm_store***. For self-managed persistence, the component implementor provides the* **ccm_load** *and* **ccm_store** *methods. Since both process and*

entity components have persistent state and container-managed persistence, the same callback interfaces can be used.

4.4 Server Programming Interfaces - Extended Components

This section defines the local interfaces used and provided by the component developer for extended components. These interfaces are grouped as in Section 4.3, ÒServer Programming Interfaces - Basic Components,Ó on page 4-21. Unless otherwise indicated, all of these interfaces are defined within the **Components** module. Extended components add interfaces in the following areas:

¥ **CCM2Context** - adds functions unique to extended components.

Each container API type has its own specialization of **CCM2Context** that we refer to as a context. The context for extended components adds accessors to persistence service~~s~~ and supports operations for managing servant lifetime policy, and creating and managing object references in conjunction with the POA.

¥ **ComponentId** - encapsulates a component identifier, which is an abstract information model used to locate the componentÕs state.

Only the **entity container API type** supports the **ComponentId** interface.

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-02 issue 3937 and ptc/01-11-03 page 62-358).

¥ ~~**Event** - offers the subset of the CORBA Notification service supported by CORBA components.~~

4.4.1 Interfaces Common to both Container API Types

This section describes the interfaces and operations provided for extended components by both **container API types** to support all categories of CORBA components.

4.4.1.1 The CCM2Context Interface

The **CCM2Context** is an **internal** interface that extends the **CCMContext** interface to provide the extended component instance with access to additional container-provided runtime services applicable to both **container API types**. These services include advanced persistence using the CORBA Persistent State service, and runtime management of component references and servants using the POA. The **CCM2Context** is defined by the following IDL:

```
typedef CosPersistentState::CatalogBase CatalogBase;
typedef CosPersistentState::TypeId TypeId;

exception PolicyMismatch { };
```

```
exception PersistenceNotAvailable { };

local interface CCM2Context : CCMContext {
    HomeRegistration get_home_registration ();
```

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-02 issue 3937 and ptc/01-11-03 page 62-358).

```
    ~~EventsNotification::Event get_event();~~
    void req_passivate () raises (PolicyMismatch);

    CatalogBase get_persistence (in TypeId catalog_type_id)
            raises (PersistenceNotAvailable);
    };
```

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**.

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 shall be raised.

get_persistence

The **get_persistence** operation provides the component access to a persistence framework provided by an implementation of the CORBA Persistence State service. It returns a **CosPersistentState::CatalogBase**, which serves as an index to the available storage homes. The **CatalogBase** is identified by its **CosPersistentState::TypeId catalog_type_id**. If the **CatalogBase** identified by **catalog_type_id** is not available on this container, the PersistenceNotAvailable exception shall be raised.

4.4.1.2 The HomeRegistration Interface

The **HomeRegistration** is an **internal** interface that 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 clientÕs find_home 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 use 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 interface HomeRegistration {
    void register_home (
        in CCMHome home_ref,
        in string home_name);
    void unregister_home (in CCMHome home_ref);
};
```

register_home

The **register_home** operation is used to register a component home with the **HomeFinder** so it can be located by a component client. The **home_ref** parameter identifies the home being registered and can be used to obtain both the **CORBA::ComponentIR::ComponentDef** (**CCMHome::get_component_def**) and the **CORBA::InterfaceDef** (**CORBA::Object::get_interface_def**) to support both **HomeFinder::find_home_by_component_type** and **HomeFinder::find_home_by_home_type**. The **home_name** parameter identifies an Interoperable Naming Service (INS) name that can be used as input to the **HomeFinder::find_home_by_name** operation. If the **home_name** parameter is NULL, no name is associated with this home so this home cannot be retrieved by name.

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_ref** parameter identifies the home being unregistered.

4.4.1.3 The ProxyHomeRegistration Interface

Because CORBA components exploit the dynamic activation features of the POA, it is possible for some component types to provide a home that is not collocated with the component instances it creates. This permits load balancing criteria to be applied in selecting the actual server and POA where this instance will be created. The **ProxyHomeRegistration** 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 **ProxyHomeRegistration** interface is defined by the following IDL:

```
exception UnknownActualHome { };
exception ProxyHomeNotSupported { };

local interface ProxyHomeRegistration : HomeRegistration {
    void register_proxy_home (
```

```
            in CCMHome rhome,
            in CCMHome ahome)
            raises (UnknownActualHome, ProxyHomeNotSupported);
};
```

register_proxy_home

The **register_proxy_home** operation is used to register a component home, not collocated with the instances that it can create, with the **HomeFinder** so the proxy home can be used by component clients. The **rhome** parameter identifies the proxy home being registered. The **ahome** parameter identifies the actual home that the **rhome** is associated with. If the actual home specified by **ahome** is not known, the UnknownActualHome exception shall be raised. If this component does not support proxy homes, the ProxyHomeNotSupported exception shall be raised. Support for proxy homes is a component implementation option.

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-02 issue 3937 and ptc/01-11-03).

~~4.4.1.4 The Event Interface~~

```
~~typedef CosNotification::EventHeader EventHeader;~~
~~typedef CosNotifyChannnelAdmin::ChannelId Channel;~~

~~exception ChannelUnavailable { };~~
~~exception InvalidSubscription { };~~
~~exception InvalidName { };~~
~~exception InvalidChannel { };~~

~~local interface LocalCookie {~~
                        ~~boolean same_as (in LocalCookie cookie);};~~
~~local interface Event {~~
                        ~~EventConsumerBase create_channel~~
                             ~~(out Channel chid)~~
                             ~~raises (ChannelUnavailable);~~
~~LocalCookie subscribe (~~
                             ~~in EventConsumerBase ecb,~~
                             ~~in Channel chid)raises (ChannelUnavailable);~~
~~void unsubscribe (in LocalCookie cookie)~~
                             ~~raises (InvalidSubscription);~~
~~EventConsumerBase obtain_channel (~~
                             ~~in string supp_name,~~
                             ~~in EventHeader hdr) raises (InvalidName);~~
~~void listen (in EventConsumerBase ecb,~~
                             ~~in string csmr_name) raises (InvalidName);~~
~~void push (in EventBase evt);~~
~~void destroy_channel (in Channel chid)raises (InvalidChannel);~~
~~};~~
```

~~EJB does not have an event API yet, but one is under development. The Java 2 Platform, Enterprise Edition (J2EE) does however have a messaging API (JMS) that supports publish/subscribe. This is an area that will need to be harmonized with EJB in the future.~~

4.4.2 Interfaces Supported by the Session Container API Type

This section describes the interfaces supported for extended components by the session container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces, which must be implemented by components deployed in this container API type.

4.4.2.1 The Session2Context Interface

The **Session2Context** is an **internal** interface that extends the **SessionContext** to provide a component instance with access to additional container-provided runtime services for the session container API type. It adds the ability to create references for components deployed in a **session** container API type. The **Session2Context** is defined by the following IDL:

```
enum BadComponentReferenceReason {
    NON_LOCAL_REFERENCE,
    NON_COMPONENT_REFERENCE,
    WRONG_CONTAINER
};

exception BadComponentReference {
    BadComponentReferenceReason reason;
};
exception IllegalState { };
```

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03 page 62-363).

```
local interface Session2Context : SessionContext, CCM2Context {
    Object create_ref (in CORBA::RepositoryId repid);
    Object create_ref_from_oid (
        in PortableServer::ObjectIdCORBA::OctetSeq oid,
        in CORBA::RepositoryId repid);
    PortableServer::ObjectId
    CORBA::OctetSeq get_oid_from_ref (in Object objref)
        raises (IllegalState, BadComponentReference);
};
```

(In the listing above, "PortableServer::ObjectId" in the create_ref_from_oid parameter and the standalone "PortableServer::ObjectId" line are struck through: ~~PortableServer::ObjectId~~.)

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. Invocations on the new object reference are delivered to the appropriate segment of the component that invokes this operation. The **RepositoryId** must match the **RepositoryId** of the component itself, one of its bases, one of its supported interfaces, or one of its facets.

create_ref_from_oid

The **create_ref_from_oid** operation is used to create a reference to be exported to clients that includes information provided by the component which it can use on subsequent operation requests. The **oid** parameter identifies the **ObjectSeq** to be encapsulated in the reference and the **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

get_oid_from_ref

The **get_oid_from_ref** operation is used by the component to extract the **oid** encapsulated in the reference. The **objref** parameter specifies the reference that contains the **oid**. This operation must be called within an operation invocation. If not, the IllegalState exception shall be raised. If the reference was not created by this container, the BadComponentReference with the **WRONG_CONTAINER** minor code is raised.

4.4.3 Interfaces Supported by the Entity Container API Type

This section describes the interfaces provided for extended components by the entity container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces, which must be implemented by components deployed in this container API type.

4.4.3.1 Component Identifiers

The **ComponentId** interface is an **internal** interface provided by the entity container API type through which the component implementation and the container exchange identity information, referred to as *component identifiers*. The **ComponentId** interface encapsulates a component identifier, which is an abstract information model. The **ComponentId** interface is used in the following ways:

- Component implementations (usually home executor implementations) create component identifiers to describe new components, and to create object references that encapsulate the provided description. The **Entity2Context** interface acts as a factory for component identifiers and as the factory for object references.
- The container encodes the information encapsulated by the component identifier in the object identifier value it uses internally to create the object reference on the encapsulated POA. The encoding is not specified, since a containerÕs choice of encoding does not affect interoperability or portability.
- While dispatching an incoming request, the container extracts and decodes the component identifier from the **ObjectId**. The extracted component identifier is made available to the component executor through the context before the request is dispatched to the component.

- When the container invokes **ccm_load** in the component executor, the implementation of **ccm_load** uses the contents of the component identifier to locate and incarnate the required component state.

In the following discussions, component identifiers and component object references are sometimes used as though the terms were synonymous. Since there is a one-to-one relationship between a component identifier and an object reference created from the component identifier, this discussion occasionally uses the term Òcomponent referenceÓ to mean Òthe component reference created from the component identifier in question,Ó for the sake of brevity.

The **ComponentId** interface does not explicitly specify the state representation it encapsulates. The abstract state is implied by the interface and reflects the structure of the executor it describes (see the *CCM Implementation Framework* chapter for a complete discussion of executor structure).

A component identifier encapsulates the following information:

- A *facet identifier* value denoting the target facet of the component reference.
- A *segment identifier* value denoting the target segment of the component reference (i.e., the segment that supports the target facet).
- A sequence of *segment descriptors*.

A segment descriptor includes the following:

- A segment identifier denotes the segment being described, and
- a *state identifier* value that denotes the persistent state of the segment in some storage mechanism.

A monolithic executor is represented as a degenerate case of the generalized component identifier, where the target segment identifier is set to zero and the sequence of segment descriptors contains a single element, whose segment identifier is zero and whose state identifier denotes the persistent state of the componentÕs single segment.

The facet identifier value zero is reserved to denote the component facet; that is, the facet that supports the component equivalent interface. The segment identifier value zero is reserved to denote the segment that supports the component facet. For monolithic executors, the segment identifier values is always zero.

State identifier is an abstraction that generalizes a variety of possible state identity schemes. This specification provides a mechanism for describing state identifiers that can be extended by component implementors, allowing customization for storage mechanisms that do not support the standard persistence interfaces.

The **ComponentId** local interface and supporting constructs are defined by the following IDL:

```
typedef short SegmentId;
const SegmentId COMPONENT_SEGMENT = 0;

typedef short FacetId;
const FacetId COMPONENT_FACET = 0;
```

```
typedef sequence<octet> IdData;
typedef CosPersistentState::Pid PersistentId;

exception InvalidStateIdData {};

typedef short StateIdType;
const StateIdType PERSISTENT_ID = 0;

abstract valuetype StateIdValue {
    StateIdType get_sid_type();
    IdData get_sid_data();
};

local interface StateIdFactory {
    StateIdValue create (in IdData data) raises (InvalidStateIdData);
};

valuetype PersistentIdValue : StateIdValue {
    private PersistentId pid;
    PersistentId get_pid();
    factory init (in PersistentId pid);
};

valuetype SegmentDescr {
    private StateIdValue sid;
    private SegmentId seg;
    StateIdValue get_sid();
    SegmentId get_seg_id();
    factory init (in StateIdValue sid, in SegmentId seg);
};

typedef sequence<SegmentDescr> SegmentDescrSeq;

local interface ComponentId {
    FacetId get_target_facet();
    SegmentId get_target_segment();
    StateIdValue get_target_state_id (in StateIdFactory sid_factory)
            raises (InvalidStateIdData);
    StateIdValue get_segment_state_id (
        in SegmentId seg,
        in StateIdFactory sid_factory)
        raises (InvalidStateIdData);
    ComponentId create_with_new_target (
        in FacetId new_target_facet,
        in SegmentId new_target_segment);
    SegmentDescrSeq get_segment_descrs (
            in StateIdFactory sid_factory)
            raises (InvalidStateIdData);
};
```

4.4.3.2 StateIdValue abstract valuetype

The **StateIdValue** type is the base valuetype for concrete, storage-specific state identity values. The container interacts with state identities completely in terms of this interface. A single pre-defined concrete value type derived from **StateIdValue** is provided for **PersistentId** state identities. Component implementors, or suppliers of storage mechanisms that do not support the CORBA component persistence model can provide their own state identity types by deriving from **StateIdValue** and implementing the required behaviors properly.

get_sid_type

The **get_sid_type** operation returns a discriminator (physically, a short) that identifies the type of the state identity encapsulated by the **StateIdValue**. This specification defines the value zero (0) to denote a **Components::~~Extended::~~PersistentId** state identifier.

get_sid_data

The **get_sid_data** operation returns the encapsulated state identity expressed in a canonical form, as a sequence of octets. The implementation of the derived concrete value type is responsible for converting its encapsulated data into this form, and for supplying a factory that can construct an instance of the concrete type from an **IdData** value (a sequence of octets).

4.4.3.3 StateIdFactory Interface

StateIdFactory is the abstract base interface for factories of state identity values derived from **StateIdValue**. An implementation of **StateIdFactory** must be supplied with the implementation of a concrete state identity type. If the **IdData** octet sequence provided in the **data** parameter cannot be decoded to create a proper instance of the expected state identity concrete type, the operation raises an InvalidStateIdData exception.

create

The **create** operation constructs an instance of a concrete state identifier from the octet sequence parameter. This operation performs the inverse of the transformation performed by the **get_sid_data**.

4.4.3.4 PersistentIdValue valuetype

The **PersistentIdValue** type is a specialization of **StateIdValue** that encapsulates a **PersistentId** value for inclusion in a component identifier.

get_pid

The **get_pid** operation returns the **PersistentId** value encapsulated by the value type.

init

The initializer for **PersistentIdValue** creates an instance of the valuetype that encapsulates the **PersistentId** value passed as a parameter.

get_sid_value

The implementation of **get_sid_value** for **PersistentIdValue** performs no transformation on the encapsulated **PersistentId** value. The sequence of octets returned by **get_sid_value** is identical to the encapsulated **PersistentId** value.

4.4.3.5 SegmentDescr valuetype

The **SegmentDescr** type describes an executor segment, encapsulating a segment identifier and a state identifier. A component identifier for a segmented executor encapsulates a sequence of **SegmentDescr** instances.

get_sid

The **get_sid** operation returns the state identity value of the segment being described.

get_seg_id

The **get_seg_id** operation returns the segment identifier of the segment being described.

init

This initializer sets the value of the encapsulated segment identifier and state identifier to the values of the respective parameters.

4.4.3.6 ComponentId Interface

The **ComponentId** interface encapsulates a complete component identity. Instances of **ComponentId** can only be created by the **Entity2Context** interface, which is supplied by the container, or by duplicating an existing component identifier with a new target value, with **ComponentId::create_with_new_target**. Instances of **ComponentId** are also provided by the **EntityContext** interface in the context of a CORBA invocation. The value of the component identifier provided by the **Entity2Context** shall be identical to the component identifier value used to create the object reference on which the invocation was made. The **ComponentId** interface is a read-only interface. Once a component identifier is constructed by the **create_component_id** operation or constructed internally and provided through the **Entity2Context** interface, the value of the component identifier cannot be altered.

get_target_facet

The **get_target_facet** operation returns the facet identifier of the facet, which is the target of the component reference; that is, the target of requests made on the component reference.

get_target_segment

The **get_target_segment** operation returns the segment identifier of the target segment; that is, the segments that provides the target facet.

get_target_state_id

The **get_target_state_id** operation returns the state identifier of the target segment. The **StateIdFactory** specified in the **sid_factory** parameter is used by the implementation of **get_target_state_id** to construct the proper state identifier from the octet sequence encapsulated by the component identifier. If the state identifier of the target segment is a **PersistentIdValue**, the **sid_factory** parameter may be nil. Container implementations shall provide a default implementation of **StateIdFactory** to be used when the encapsulated state identifier value is a **PersistentIdValue**. If provided (or default) factory cannot construct a correct state identifier of the expected type from the undecoded octet sequence encapsulated by the component identifier, the operation raises an **InvalidStateIdData** exception.

get_segment_state_id

The **get_segment_state_id** operation returns the state identifier of the segment specified by the **seg** parameter. The semantics are otherwise identical to **get_target_state_id**, with respect to the meaning and use of the **sid_factory** parameter.

get_segment_descrs

The **get_segment_descrs** operation returns a sequence containing all of the segment descriptors encapsulated by the component identifier. The sequence is a copy of the encapsulated sequence. The state identifier factory in the **sid_factory** parameter (or the default) is used by the implementation of **get_segment_descrs** to construct state identifiers of the appropriate concrete subtype of **StateIdValue**. If provided (or default) factory cannot construct a correct state identifier of the expected type from the undecoded octet sequence encapsulated by the component identifier, the operation raises an InvalidStateIdData exception.

create_with_new_target

The **create_with_new_target** operation creates a new component identifier that is identical to the target component identifier, except that the target facet and target segment values are replaced with the values of the **new_target_facet** and **new_target_segment** parameters, respectively.

This operation is intended primarily to be used in implementing navigation *operations.*

4.4.3.7 The Entity2Context Interface

The **Entity2Context** is an **internal** interface that extends the **EntityContext** interface to provide the extended component with access to additional container-provided runtime services for managing object references and advanced persistence. Object references for components deployed in an entity container API type can choose to use the CORBA Persistent State service or some user defined persistence mechanism. The **ComponentId** interface (defined in Section 4.4.3.6, ÒComponentId Interface,Ó on page 4-42) encapsulates this distinction when a reference is to be used. The **Entity2Context** is defined by the following IDL.

```
exception BadComponentReference {
    BadComponentReferenceReason reason; };
exception IllegalState { };

local interface Entity2Context : EntityContext, CCM2Context {
    ComponentId get_component_id ()
        raises (IllegalState);
    ComponentId create_component_id (
        in FacetId target_facet,
        in SegmentId target_segment,
        in SegmentDescrSeq seq_descrs);
    ComponentId create_monolithic_component_id (
        in FacetId target_facet,
        in StateIdValue sid);
    Object create_ref_from_cid (
        in CORBA::RepositoryId repid,
        in ComponentId cid);
    ComponentId get_cid_from_ref (
        in Object objref) raises (BadComponentReference);
};
```

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 that can be used to access the componentÕs persistence state. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

create_component_id

The **create_component_id** operation creates a component identifier value, initializing it with the values specified in the parameters. The **target_facet** parameter contains the facet identifier of the target facet, the **target_segment** parameter contains the segment identifier of the target segment, and the **seq_descrs** parameter contains a sequence of segment descriptors describing all of the segments that constitute the component executor.

create_monolithic_component_id

The **create_monolithic_component_id** operation provides a simplified signature for creating a component identifier value for monolithic executors, which have a single segment. The **target_facet** parameter contains the facet identifier of the target facet, and the **sid** parameter contains the state identifier for the single executor segment. The target segment identifier encapsulated by the component identifier is set to zero, and the sequence of segment descriptors encapsulated by the component identifier has a single element, initialized with segment identifier value zero, and state identifier value specified by the **sid** parameter.

create_ref_from_cid

The **create_ref_from_cid** operation is used by a component factory to create an object reference that 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 **EntityComponent callback** operations are invoked. The **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

get_cid_from_ref

The **get_cid_from_ref** operation is used by a persistent component to retrieve the **ComponentId** encapsulated in the reference (**objref**). 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**).

> *The* **ComponentId** *structure is dependent on the home implementation and the container, in particular, its implementation of the* **Entity2Context** *interface. It is likely that a* **ComponentId** *created by one container will not be understandable by another, hence the possibility of the* **WRONG_CONTAINER** *exception.*

4.5 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 IDL extensions has been described previously (see the *Component Model* chapter). This section focuses on the use of standard CORBA by the client who wishes to communicate with a CORBA component implemented in a **Component Server**. It enables a CORBA client, which is not itself a CORBA component, to communicate with a CORBA component.

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 that allow the client to obtain references to an application interface which the component implements.

From the clientÕs perspective, the home supports two design patterns - factories for creating new objects and finders for existing objects. These are distinguished by the presence of a **primarykey** parameter in the home IDL.

- If a **primarykey** is defined, the home supports both factories and 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; for example, 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; for example, looking up a name in a Naming or Trader service, searching for a particular type of factory using a factory finder, etc.

4.5.1 Component-aware Clients

Clients that are defined using the IDL extensions in the *Component Model* chapter are referred to as **component-aware** clients. Such clients can avail themselves of the unique features of CORBA components that 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 interacts with a component through one or more CORBA interfaces:

- The equivalent interface implied by the **component** IDL declaration.
- Zero or more supported interfaces declared on the **component** specification.
- Zero or more interfaces defined by the **provides** clauses in the **component** definition.
- The home interface that 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** that provides the initial set of object references.

4.5.1.1 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 (**ÒNameServiceÓ**)

¥ Transaction Current (**ÒTransactionCurrentÓ**)

¥ Security Current (**ÒSecurityCurrentÓ**)

¥ Notification Service (**ÒNotificationServiceÓ**)

¥ Interface Repository (**ÒInterfaceRepositoryÓ**) for DII clients

¥ Home Finder (**ÒComponentHomeFinder**Ó)

The client uses ComponentHomeFinder (defined in Section 1.8, ÒHome Finders,Ó on page 1-42) to obtain a reference to the **HomeFinder** interface.

4.5.1.2 *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** that 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 (**CCMHome**) 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:

```
// 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);
```

4.5.1.3 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. ClientÕs may use the **HomeFinder** as described in Section 1.8, ÒHome Finders,Ó on page 1-42 to locate the componentÕs home or they may use CORBA naming to look up a specific instance of the home by symbolic name.

A client using the finder design pattern uses the **CosNaming::NamingContext** interface to look up 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:

```
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);
```

4.5.1.4 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 **ÒTransactionCurrentÓ**. This permits the client to define the boundaries of the transaction; that is, how many operations will be invoked within the scope of the clientÕs 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 service 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();
```

4.5.1.5 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 that will be used to authorize subsequent requests. The credentials must be set in a way that 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 **ÒSecurityCurrentÓ**. 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);
```

4.5.1.6 Events

Component-aware clients wishing to **emit** or **consume** events use the component APIs defined in the *Component Model* chapter. Alternatively, they may use CORBA notification directly and conform to the subset supported by CORBA components (see Section 4.5.2.6, ÒEvents,Ó on page 4-51 for details).

4.5.2 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.

4.5.2.1 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**.

4.5.2.2 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.
- 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 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 ÒbestÓ factory for the client request based on its knowledge of workload, etc.*

A **FactoryFinder** will return an **Object**. A component-unaware client may expect to narrow this to **CosLifeCycle::GenericFactory** and use the generic create operation. For this reason, we allow the default creation operation on home to return a **GenericFactory** interface. This is fully described in Section 1.7, ÒHomes,Ó on page 1-32.

- 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. A supported interface must be used by the client to invoke the componentÕs application operations. Provided interfaces cannot be located using the factory design pattern.

4.5.2.3 Finder Design Pattern

A component-unaware client can use CORBA naming to locate an existing **entity** component. Unlike the factory design pattern, the name to be looked up by the client can be either a supported interface or any of the provided interfaces. The following techniques can be used:

- A symbolic name associated with the componentÕs home can be looked up in a Naming service to make an invocation of the finder operations.
- 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.

4.5.2.4 Transactions

This is the same as component-aware clients (See Section 4.5.1.4, ÒTransactions,Ó on page 4-48). 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.

4.5.2.5 Security

This is the same as component-aware clients (See Section 4.5.1.5, ÒSecurity,Ó on page 4-49).

4.5.2.6 Events

Component-unaware clients wishing to **emit** or **consume** events must use the equivalent CORBA notification interfaces and stay within the subset supported by CORBA components (see Section 4.2.8, ÒEvents,Ó on page 4-12 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(...);
```

Integrating with Enterprise JavaBeans

5

This chapter describes the integration of CORBA components with Enterprise JavaBeans.

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒIntroductionÓ	5-1
ÒEnterprise JavaBeans Compatibility Objectives and RequirementsÓ	5-3
ÒCORBA Component Views for EJBsÓ	5-4
ÒEJB views for CORBA ComponentsÓ	5-14
ÒCompliance with the Interoperability of Integration ViewsÓ	5-22
ÒComparing CCM and EJBÓ	5-23

5.1 Introduction

This chapter describes how an Enterprise JavaBeans (EJB) component can be used by CORBA clients, including CORBA components. The EJB will have a CORBA component style remote interface that is described by CORBA IDL (including the component extensions).

This chapter also describes how a CORBA component can be used by a Java client, including an Enterprise JavaBeans component. The CORBA component will have an EJB style remote interface that is defined following the Enterprise JavaBeans specification.

The concepts in this chapter follow in the same prescription for interworking as laid out in the *Common Object Request Broker Architecture (CORBA)* specification, *Interworking Architecture* chapter where it is discussed as follows:

How interworking can be practically achieved is illustrated in an Interworking Model, shown in Figure 5-1. It shows how an object in Object System B can be mapped and represented to a client in Object System A. From now on, this will be called a B/A mapping. For example, mapping a CORBA Component Model object to be visible to an EJB client is a CCM/EJB mapping.

On the left is a client in object system A, that wants to send a request to a target object in system B, on the right. We refer to the entire conceptual entity that provides the mapping as a bridge. The goal is to map and deliver any request from the client transparently to the target.

To do so, we first provide an object in system A called a View. The View is an object in system A that presents the identity and interface of the target in system B mapped to the vernacular of system A, and is described as an A View of a B target. The View exposes an interface, called the View Interface, which is isomorphic to the targetÕs interface in system B. The methods of the View Interface convert requests from system A clients into requests on the targetÕs interface in system B. The View is a component of the bridge. A bridge may be composed of many Views.

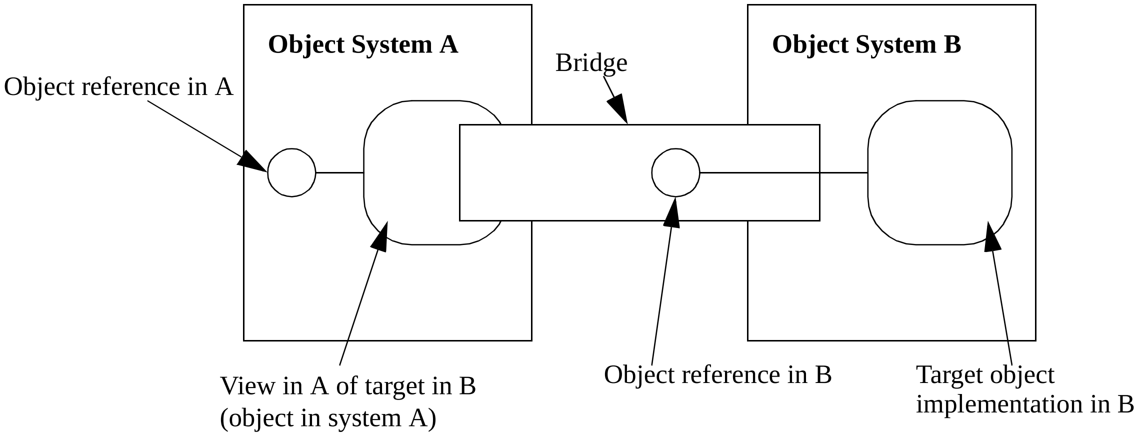


Figure 5-1 B/A Interworking Model

The bridge maps interface and identify forms between different object systems. Conceptually, the bridge holds a reference in B for the target (although this is not physically required). The bridge must provide a point of rendezvous between A and B, and may be implemented using any mechanism that permits communication between the two systems (IPC, RPC, network, shared memory, and so forth) sufficient to preserve all relevant object semantics.

The client treats the View as though it is the real object in system A, and makes the request in the vernacular request form of system A. The request is translated into the vernacular of object system B, and delivered to the target object. The net effect is that a request made on an interface in A is transparently delivered to the intended instance in B.

The Interworking Model works in either direction. For example, if system A is EJB, and system B is CCM, then the View is called the EJB View of the CCM target. The EJB View presents the targetÕs interface to the EJB client. Similarly if system A is CCM and system B is EJB, then the View is called the CCM View of the EJB target. The CCM View presents the targetÕs interface to the CCM client.

5.2 Enterprise JavaBeans Compatibility Objectives and Requirements

The objective is to allow the creation of distributed applications that mix CORBA components running in CORBA component servers with EJB components running in an EJB technology-based server. This objective allows a developer to create an application by reusing existing components of either kind.

This requires development time and runtime translations between the CORBA component and EJB domains provided by mediated bridges. It also requires that:

- A CORBA component view for an EJB comply with the EJB to CORBA mapping specification. In particular, this requires that:
  - An EJB definition be mapped to a CORBA component definition following the Java Language to IDL mapping plus the extensions to that mapping that are specified in this chapter.
  - Value objects of one kind (e.g., Keys for EJB) have counterpart value objects of the other kind.
  - CORBA components accessible via **CosNaming** have their EJB views accessible via **JNDI**, and vice versa.
- An EJB view for a CORBA component comply with the EJB specification.

An application is to be built using both EJB and CORBA components deployed in their respective containers. At component development time, EJB components are originally defined in Java and CORBA components are originally defined in IDL. When applications are assembled using both, the application assembly environment will most commonly dictate which model these components must present to developers. During application assembly, developers construct clients (which themselves may be components) that make use of components in the way most natural to the particular environment. Thus in a CORBA environment clients will expect to make use of both the CCM model and the EJB model as CORBA components, and in an EJB environment, clients will expect to make use of both kinds as enterprise beans. All four combinations of clients and components are illustrated in Figure 5-2.

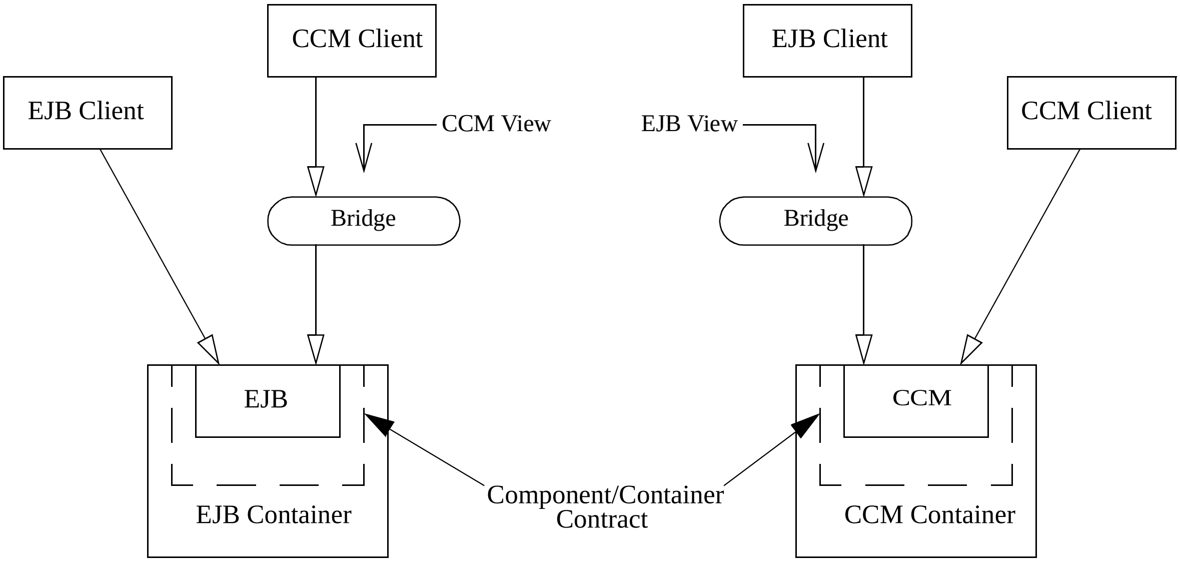


Figure 5-2 Interoperation in a mixed environment

In this scenario, components of one kind are made accessible to clients of another by way of two mechanisms: generation of bindings at development time and method translation at runtime. Thus, the containers provide an EJB view of a CORBA component and a CCM view of an EJB.

For application developers in a CORBA environment, EJBs specified in Java are mapped to CORBA IDL for use by CCM clients, and at runtime client calls on CCM methods are translated by a bridge into EJB methods. In effect, the EJBs *are* CORBA components.

For application developers in an EJB environment, CORBA components specified in IDL are mapped to Java interfaces for use by EJB clients, and at runtime client calls on EJB methods are translated by a bridge into CCM methods. In effect, the CORBA components *are* EJBs.

5.3 CORBA Component Views for EJBs

This kind of view allows a CORBA client -- either a CORBA component or any piece of code that uses CORBA, and either component-aware or not -- to access an EJB as a CORBA component. To do this, two things are needed:

1. A mapping of the definition of the existing EJB into the definition of a CORBA component. This mapping takes an EJBÕs RMI remote interface and home interface and produces an equivalent CORBA component definition.
2. A translation, at run-time, of CORBA component requests performed by a CORBA client into EJB requests. This translation can be performed in terms of either straight delegation, or as an interpretation of a CORBA client request in terms of EJB requests.

5.3.1 Mapping of EJB to Component IDL definitions

An EJB definition includes the following EJB interfaces:

- EJB home interface - This interface extends the pre-defined **EJBHome** interface.
- EJB remote interface - This interface extends the pre-defined **EJBObject** interface.

Thus, for the purposes of this chapter, at least these EJB interfaces must be mapped into IDL in order to obtain a CORBA component definition of a view that a CORBA client can use to make requests on an existing EJB. An EJB home interface definition maps into a CORBA componentÕs home definition, whose implied IDL inherits from **CCMHome**. This means that **EJBHome** is mapped into **CCMHome**. Likewise, an EJB remote interface definition maps into a basic CORBA component definition, whose implied IDL inherits from **CCMObject**. This means that **EJBObject** is mapped into **CCMObject**.

In addition, **EJBHome** and **EJBObject** make use of the following pre-defined EJB interfaces:

- **HomeHandle**
- **Handle**
- **EJBMetaData**

Handles are an EJB concept that has no direct counterpart in CORBA components. Thus, **HomeHandle** and **Handle** are not directly mapped into equivalent IDL.

> *Notice that although Interoperable Object References (IORs) and the ORB provided operations that manipulate them (***string_to_object*** and* **object_to_string***) are conceptually similar to Handles, there are enough differences between IORs and Handles to preclude a mapping from Handles to IORs.*

Meta data is available to a CORBA client but not in the same form as that provided by **EJBMetaData**. Given that an EJB maps into a CORBA component, whose definition produces the meta data that a CORBA client expects, mapping **EJBMetaData** into equivalent IDL is not required.

5.3.1.1 Java Language to IDL Mapping

The reader is assumed to be familiar with the specification for the Java to IDL mapping, whose major aspects are repeated here for convenience.

- A Java interface is an RMI/IDL remote interface if it at least extends `java.rmi.Remote` and all of its methods throw `java.rmi.RemoteException`.
- 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 that should be mapped to IDL operations, and the method signatures must meet the requirements of the mapping.
  - Objects that inherit from `java.io.Externalizable` or that implement `writeObject` are understood to perform custom marshalling and the corresponding custom marshallers must be created for the CORBA valuetype.
- Arrays are mapped to ÒboxedÓ 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.

5.3.1.2 EJB to IDL mapping

In general, the CORBA component that results from mapping an EJB will support an interface that is the Java to IDL map of the Remote interface of the EJB. The mapping rules are as follows.

Mapping the Remote Interface

- An EJBÕs remote interface maps to a definition of a basic CORBA component that supports the default interface. The form of the CORBA component definition is **component *XXX* supports *XXX*Default**.
- An EJBÕs remote interface declaration is used to create a **supports** declaration and the corresponding IDL for the primary interface of the CORBA component that the EJB maps to. The identifier of this supported interface on the component is

XXX*Default**, where ***XXX is the name of the EJB remote interface. This generated interface is referred to as the *Default* interface of the component that the given EJB maps to.

- Each operation on the Remote interface is mapped under Java to IDL to an equivalent operation on the ***XXX*Default** interface.
- Each pair of **get*XXX*** and **set*XXX*** methods in the EJB remote interface will be mapped to IDL attributes in the component definition itself. Any exceptions thrown by a **get*XXX*** method is mapped to an exception in the **getraises** clause of the mapped IDL attribute. Likewise, any exception thrown by a **set*XXX*** method is mapped to an exception in the **setraises** clause of the mapped IDL attribute. The actual definitions of the exceptions thrown are mapped following the Java to IDL rules.

Mapping the Home Interface

- An EJBÕs home interface maps to a definition of a CORBA component home. The form of the CORBA component home definition is **home *YYY* manages *XXX***, where ***YYY*** is the name of the EJB home interface. Mapping an EJB home into a CORBA component home requires the existence of meta data that links the EJB home to the EJB that it hosts. These meta data are obtained from the EJBÕs deployment descriptor. Thus ***XXX*** is the name of the EJB that the EJB home hosts, as it is given in the EJB deployment descriptor.
- The EJB home methods called **create** are mapped into home **factory** declarations in IDL. The actual names of each of the **factory** operations are produced following the rules for mapping Java names to IDL names in the Java to IDL specification. The Java parameters of the operation are mapped to their corresponding IDL types and names as defined by Java to IDL.
- An EJB Primary Key class is mapped to a CORBA **valuetype** using the mapping rules in Java to IDL. This **valuetype** will be declared in the IDL for the CORBA component home as the primary key **valuetype** for the component. The key **valuetype** will inherit from **Components::PrimaryKeyBase**. If an EJB home uses a primary key, then the form of the CORBA component home definition is **home *YYY* manages *XXX* primarykey *KKK***, where ***KKK*** is the name of the valuetype that the EJB primary key class maps to.
- The EJB home operation named **findByPrimaryKey** is mapped into the **find_by_primary_key(in <key-type> key)** operation on the componentÕs implicit home interface.
- Finder and Creator EJB operations that return an RMI style object reference are mapped into Component IDL operations that return a CORBA Component Object Reference to ***XXX***.
- EJB home operations prefixed **find** whose return type is the type of the EJB hosted by the EJB home are mapped into component home **finder** operations in IDL. The actual names of each of the **finder** operations are produced following the rules for mapping Java names to IDL names in the Java to IDL specification. The Java parameters of the operation are mapped to their corresponding IDL types and names as defined by Java to IDL.

- Finder EJB operations that return a Java Enumeration are mapped into CORBA component operations that return a value of type **Enumeration**. This value type is declared as:

  ```
  module Components {
      abstract valuetype Enumeration {
          boolean has_more_elements();
          CCMObject 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.

  A concrete specialization of this abstract value type must be provided. This specialization has the form:

  ```
  module Components {
      typedef sequence<CCMObject> CCMObjectSeq;
      valuetype DefaultEnumeration : Enumeration {
          private CCMObjectSeq objects;
      };
  };
  ```

  Any implementation of **DefaultEnumeration**, in any language, must provide implementations for the two **Enumeration** methods. Any client ORB that supports the interoperable bridge has to provide an implementation that knows how to read **DefaultEnumeration** from the wire and to use that information to provide a local implementation of these two methods. Any EJB container that supports the CCM-EJB bridge has to provide an implementation that knows how to construct itself from a **java.util.Enumeration** and then write itself to the wire as a **DefaultEnumeration**.

- In order for an EJB home definition that defines **findByPrimaryKey** to be successfully mapped onto a CORBA component home definition, it must define a `create` method that takes the primary key of the hosted EJB as its sole argument and returns an instance of the hosted EJB. This create method is mapped to **create(in <key-type> key)** on the CORBA component implicit home interface.

Mapping standard exceptions

The EJB exceptions **FinderException**, **CreateException**, **DuplicateKeyException**, and **RemoveException** thrown by methods to find, create, and remove an EJB are always mapped to the CCM exceptions **Components::FinderFailure**, **Components::CreateFailure**, **Components::DuplicateKeyValue** and **Components::RemoveFailure**, respectively.

5.3.2 Translation of CORBA Component requests into EJB requests

A CORBA client that uses a CORBA component view on an EJB expects to be able to perform CORBA component requests on such a view. These requests need to be translated into EJB requests at run-time. This translation can be performed at the client-side, server-side, or a combination of the two. Table 5-1 lists the CORBA component operations that a CORBA client can perform requests on by interface, and it lists the corresponding EJB methods that these requests translate into, also by interface.

Table 5-1 Translation of CCM operation requests into EJB method requests

CCM Interface	**Operation called by client**	**EJB interface**	**Method invoked by bridge**
CCMHome	ComponentDef get_component_def ();	EJBHome	EJBMetaData getEJBMetaData () throws RemoteException;
	CORBA::IRObject get_home_def ();		EJBMetaData getEJBMetaData() throws RemoteException;
	void remove_component (in CCMObject comp) raises (RemoveFailure);		void remove (Handle handle) throws RemoveException, RemoteException;
<home-name>Explicit	<name> createXXX (<arg-list>) raises (CreateFailure, DuplicateKeyValue, InvalidKey);	<home-name>	<name> create (<arg-list>) throws CreateException, DuplicateKeyException;
	<name> findXXX (<arg-list>) raises (FinderFailure, <exceptions>);		<name> findXXX (<arg-list>) throws <exceptions>;
<home-name>Implicit	<name> create (in <key-type> key) raises (CreateFailure, DuplicateKeyValue, InvalidKey);		<name> create (Object primaryKey) throws CreateException, DuplicateKeyException;
	<name> find_by_primary_key (in <key-type> key) raises (FinderFailure, UnknownKeyValue, InvalidKey);		<name> findByPrimaryKey (<key-type> key) throws FinderException, ObjectNotFoundException;
	void remove (in <key-type> key) raises (RemoveFailure, UnknownKeyValue, InvalidKey);	EJBHome	void remove (Object primaryKey) throws RemoveException, RemoteException;
	<key_type> get_primary_key (in <name> comp);	EJBObject	Object getPrimaryKey () throws RemoteException;
CCMObject	ComponentDef get_component_def ();	EJBHome	EJBMetaData getEJBMetaData () throws RemoteException;
	CCMHome get_ccm_home ();	EJBObject	EJBHome getEJBHome() throws RemoteException;

Table 5-1 Translation of CCM operation requests into EJB method requests

CCM Interface	**Operation called by client**	**EJB interface**	**Method invoked by bridge**
CCMObject (continued)	PrimaryKeyBase get_primary_key ();	EJBObject	Object getPrimaryKey () throws RemoteException;
	void remove() raises (RemoveFailure);		void remove () throws RemoveException, RemoteException;
	void configuration_complete () raises (InvalidConfiguration);		Translation performed by bridge is to raise the NO_IMPLEMENT exception
<name>	<res-type> <operation> (<arg-list>) raises (<exceptions>);	<name>	<res-type> <operation> (<arg-list>) throws <exceptions>;
	<res-type> getXXX () throws <exceptions>;		<res-type> getXXX () throws <exceptions>;
	void setXXX (<arg-list>) throws <exceptions>;		void setXXX (<arg-list>) throws <exceptions>;

Notice that a CORBA client may use operations on object references such as **string_to_object** and **object_to_string** that may be considered as analogous to EJB **Handle** methods. However, these operations are not seen by the bridge since they are performed on the ORB and thus no translation for these operations on the part of the bridge is required.

The following restrictions apply:

- **create (in <key_type> key)** on the component implicit home interface can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **remove (in <*key_type*> key)** on the component implicit home interface can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **get_primary_key** on the component implicit home and on **CCMObject** can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **configuration_complete** on **CCMObject** is not translated by the bridge, a request on this operation by a CORBA client raises the NO_IMPLEMENT exception.

5.3.3 Interoperability of the View

As stated in Section 5.3.2, ÒTranslation of CORBA Component requests into EJB requests,Ó on page 5-9, translation of CORBA Component requests into EJB requests can happen at either the client-side, the server-side, or a combination of the two.

However, in order to provide interoperability of implementations of CORBA component views of EJBs, a minimal number of translation points must be performed and they must be performed at an explicitly defined location: either the client-side or the server-side. For the implementation of a CORBA component view of an EJB, and for an EJB home interface, the translation points are as follows.

Translation of specific method names

The following methods shall translate their names as indicated.

Table 5-2 Translation of specific method names

CCM Interface	**Method name**	**EJB Interface**	**Translation**
CCMHome	get_component_def	EJBHome	getEJBMetaData
	remove_component		remove
<name>Implicit	find_by_primary_key	<name>	findByPrimaryKey
	remove		remove__java_lang_Object
	create		create__java_lang_Object
	get_primary_key	EJBObject	getPrimaryKey
CCMObject	get_ccm_home	EJBObject	getEJBHome
	get_primary_key		getPrimaryKey

Handling of standard exceptions

The following exceptions, caught by the indicated methods, shall be translated as indicated before raising them to their CORBA clients.

Table 5-3 Handling of standard exceptions

CCM Interface	**Method name**	**Exception caught**	**Translation**
CCMHome	get_component_def	RemoteException	CORBA::UNKNOWN
	remove_component	RemoveException RemoteException	Components::RemoveFailure CORBA::UNKNOWN
<name>Implicit	create	DuplicateKeyException CreateException	Components::DuplicateKeyValue Components::CreateFailure
	find_by_primary_key	ObjectNotFoundException FinderException	Components::UnknownKeyValue Components::FinderFailure
	remove	RemoveException RemoteException	Components::RemoveFailure CORBA::UNKNOWN
	get_primary_key	RemoteException	CORBA::UNKNOWN
CCMObject	get_ccm_home	RemoteException	CORBA::UNKNOWN
	get_primary_key	RemoteException	CORBA::UNKNOWN

Table 5-3 Handling of standard exceptions

CCM Interface	Method name	Exception caught	Translation
	remove	RemoveException RemoteException	Components::RemoveFailure CORBA::UNKNOWN

Note Ð RemoteException is translated into **CORBA::UNKNOWN** system exception according to rules defined in formal/01-06-07, section 1.4.7.

Handling of a primary key parameter

The methods **create**, **find_by_primary_key** and **remove**, defined by **<home>Implicit** shall translate the primary key valuetype they get as input parameter to a **CORBA::Any** equivalent. Likewise, the method **get_primary_key** defined by **<home>Implicit** shall translate the **CORBA::Any** value of the primary key it gets as a result from its request into an equivalent primary key valuetype before returning it.

The method **get_primary_key**, defined by **CCMObject**, shall translate the **CORBA::Any** value of the primary key it gets as a result from its request into an equivalent **Components::PrimaryKeyBase** valuetype before returning it.

5.3.4 CORBA Component view 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.

```
package example;

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.

```
<ejb-jar>
   <session>
      <description>
      </description>
      <ejb-name> CustomerInquiry </ejb-name>
      <home> example.CustomerInquiryHome </home>
      <remote> example.CustomerInquiry </remote>
      <ejb-class> example.CustomerInquiryBean </ejb-class>
      <session-type> Stateful </session-type>
   </session>
</ejb-jar>
```

The EJB is a session bean, 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;
   public ::CORBA::WStringValue custName;
   public ::CORBA::WStringValue custAddr;
};

valuetype CustBal {
   public long custNo;
   public 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 CustomerInquiryHome manages CustomerInquiry {
        factory create();
};
```

5.4 EJB views for CORBA Components

This kind of view allows a Java client -- either an EJB or any other piece of Java code -- to access a CORBA component as an EJB. To do this, two things are needed:

¥ A mapping of the Component IDL definition of a CORBA component into an EJB definition. This mapping only considers that portion of the Component IDL language that has a counterpart in the EJB specification language and it ignores the rest. Notice that ÒThe home and remote interfaces of the enterprise bean's client view are defined as Java RMI interfaces. This allows the Container to implement the home and remote interfaces as distributed objects.Ó One implication of this is that the signatures on methods on an EJB's remote interface can only include parameters with **in** semantics. That is, **out** and **inout** semantics for parameters is not allowed. As a consequence, the **out** and **inout** qualifiers for parameters in IDL interface method definitions are not included in the portion of Component IDL that can be mapped to an EJB definition.

  Note however that a Java client does not have to use an EJB view in order to access a CCM. Any Java client can access a CCM directly via its IDL interface using a standard Java ORB, such as the one built into the JDK. This provides full access to all aspects of the CCM. Since the EJB view is derived using the IDL to Java mapping rules, the Java IDL interface is identical to the EJB view for all business operations. The only differences are in the operations mentioned in Table 5-4 on page 5-18 have slightly different names and signatures.

¥ A translation, at run-time, of EJB requests performed by a Java client into CORBA component requests.

5.4.1 Mapping of Component IDL to Enterprise JavaBeans specifications

The portion of the Component extensions to the IDL language that can be mapped to the EJB specification language is denoted by the following subset of the Component extensions to IDL grammar.

```
<component_dcl> ::= <component_header> Ò{Ó <component_body> Ò}Ó
```

```
<component_header> ::= ÒcomponentÓ <identifier> [
                    <supported_interface_spec> ]

<supported_interface_spec> ::= ÒsupportsÓ <scoped_name> { Ò,Ó
                    <scoped_name> }*

<component_body> ::= <component_export>*

<component_export> ::= <attr_dcl> Ò;Ó

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

<readonly_attr_spec> ::= ÒreadonlyÓ ÒattributeÓ <param_type_spec>
                    <readonly_attr_declarator>

<readonly_attr_declarator> ::= <simple_declarator> <raises_expr> |
                    <simple_declarator> { Ò,Ó <simple_declarator> }*

<attr_spec> ::= ÒattributeÓ <param_type_spec> <attr_declarator>

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

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

<get_excep_expr> ::= ÒgetraisesÓ <exception_list>

<set_excep_expr> ::= ÒsetraisesÓ <exception_list>

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

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

<home_header> ::= ÒhomeÓ <identifier> ÒmanagesÓ <scoped_name> [
                    <primary_key_spec> ]

<primary_key_spec> ::= ÒprimarykeyÓ <scoped_name>

<home_body> ::= Ò{Ó <home_export>* Ò}Ó

<home_export> ::= <factory_dcl> Ò;Ó | <finder_dcl> Ò;Ó

<factory_dcl> ::= ÒfactoryÓ <identifier> Ò(Ò [ <init_param_decls> ] Ò)Ó [
                    <raises_expr> ]

<finder_dcl> ::= ÒfinderÓ <identifier> Ò(Ò [ <init_param_decls> ] Ò)Ó [
                    <raises_expr> ]
```

The rules for mapping a CORBA component definition into an EJB definition are defined in the following sections. Where appropriate, these rules rely on the standard IDL to Java mapping.

Mapping the component definition

¥ A basic CORBA component definition is mapped to an EJB remote interface definition.

¥ The name of the EJB remote interface is the name of the basic CORBA component in the Component IDL definition.

¥ For each operation defined in each interface that the CORBA component **supports**, a method definition will be included in the EJB remote interface that the CORBA component maps to. That is, the EJB to which the basic CORBA component maps defines all the supported operations defined by the basic CORBA component.

¥ The signatures of the CORBA component operations are mapped to signatures of EJB remote interface methods following the IDL to Java mapping rules. Only signatures whose parameters have an **in** qualifier are allowed. Signatures that include parameters with **out** or **inout** qualifiers shall be signaled as an error.

¥ For each attribute ***XXX*** that the CORBA component defines, the corresponding EJB remote interface defines a pair of **get*XXX*** and **set*XXX*** methods, where ***XXX*** is the name of the given attribute. If the attribute definition includes a **getraises** exception clause, then the corresponding **get*XXX*** method definition in the EJB remote interface will include a throws exception clause. Likewise, if the attribute definition includes a **setraises** exception clause, then the corresponding **set*XXX*** method definition in the EJB remote interface will include a throws exception clause.

¥ Exceptions raised by CORBA component definition operations and attributes are mapped to exceptions thrown by EJB method definitions using the standard IDL to Java mapping rules.

Mapping the Component Home definition

¥ A CORBA componentÕs home definition is mapped to an EJB homeÕs remote interface definition. That is a definition of the form **home XXX manages YYY [primarykey KKK]** is mapped to an EJB home interface with name **XXX**.

¥ The methods defined by the EJB home remote interface include the implicit as well as the explicit methods of the CORBA componentÕs home definition.

¥ Implicit CORBA component home operations are mapped to EJB home remote interface methods as follows:

  ¥ `<component_type> create (in <key_type> key) raises (Components::CreateFailure, Components::DuplicateKeyValue, Components::InvalidKey);` maps to `<component_type> create (<key_type> key) throws DuplicateKeyException, CreateException`.

  - `<component_type> find_by_primary_key (in <key_type> key) raises (Components::FinderFailure, Components::UnknownKeyValue, Components::InvalidKey);` maps to `<component_type> findByPrimaryKey( <key_type> key ) throws ObjectNotFoundException, FinderException`.
  - `void remove (in <key_type> key) raises (Components::RemoveFailure, Components::UnknownKeyValue, Components::InvalidKey);` maps to the remove by key method defined in **EJBHome**.
  - `<key_type> get_primary_key (in <component_type> comp);` has no counterpart in an EJB home definition. Given that **EJBObject** already defines `getPrimaryKey`, it is not necessary to map `get_primary_key` on the implicit home to an EJB home operation.
- Explicit CORBA component basic home operations are mapped to EJB home remote interface methods as follows:
  - A **factory** operation maps to an overloaded **create** method with the corresponding arguments and exceptions.
  - A **finder** operation maps to a **find<identifier>** method with the corresponding arguments and exceptions, where **<identifier>** is the name of the **finder** operation.
  - The signatures of **factory** and **finder** operations are mapped to signatures of EJB home interface methods following the IDL to Java mapping rules.
- A **valuetype** that is used to define the primary key of a CORBA component home is mapped to a Java class under the rules of the standard IDL to Java mapping. In addition, such a Java class is defined to extend **`java.io.Serializable`**.

Mapping standard exceptions

The CCM exceptions **Components::FinderFailure**, **Components::CreateFailure**, **Components::DuplicateKeyValue** and **Components::RemoveFailure** raised by methods to find, create and remove a CORBA component are always mapped to the EJB exceptions **FinderException**, **CreateException**, **DuplicateKeyException** and **RemoveException**, respectively.

5.4.2 Translation of EJB requests into CORBA Component Requests

A Java client that uses an EJB view on a CORBA component expects to be able to perform EJB requests on such a view. These requests need to be translated into CORBA component requests at run-time. This translation can be performed at the client-side, the server-side, or a combination of the two. Table 5-4 lists the EJB methods that a Java client can perform requests on by interface, and it lists the corresponding CORBA component operations that these requests translate into, also by interface.

Table 5-4 Translation of EJB method requests into CCM operation requests

EJB Interface	Method called by client	CCM interface	Operation called by bridge
EJBHome	EJBMetaData getEJBMetadata () throws RemoteException;	CCMHome	Translation performed by bridge does not call a CCM standard operation
	void remove (Handle handle) throws RemoveException, RemoteException;		void remove_component (in CCMObject comp) raises (RemoveFailure);
	void remove (Object primaryKey) throws RemoveException, RemoteException;	<home-name>Implicit	void remove (in <key-type> key) raises (RemoveFailure, UnknownKeyValue, InvalidKey);
	HomeHandle getHomeHandle () throws RemoteException;		Translation performed by bridge does not call a CCM standard operation
<home-name>	<name> create (<arg-list>) throws CreateException, DuplicateKeyException;	<home-name>Explicit	<name> createXXX (<arg-list>) raises (CreateFailure, DuplicateKeyValue, InvalidKey);
	<name> findByXXX (<arg-list>) throws <exceptions>;		<name> findXXX (<arg-list>) raises (FinderFailure, <exceptions>);
	<name> findByPrimaryKey (<key-type> key) throws FinderException, ObjectNotFoundException;	<home-name>Implicit	<name> find_by_primary_key (in <key-type> key) raises (FinderFailure, UnknownKeyValue, InvalidKey);
EJBObject	EJBHome getEJBHome () throws RemoteException;	CCMObject	CCMHome get_ccm_home ();
	Object getPrimaryKey () throws RemoteException;		PrimaryKeyBase get_primary_key ();
	void remove () throws RemoveException, RemoteException;		void remove () raises (RemoveFailure);
	boolean isIdentical (EJBObject object) throws RemoteException;	CORBA::Object	boolean is_equivalent ();
	Handle getHandle () throws RemoteException;		Translation performed by bridge does not call a CCM standard operation.
<name>	<res-type> <operation> (<arg-list>) throws <exceptions>;	<name>	<res-type> <operation> (<arg-list>) raises (<exceptions>);
	<res-type> getXXX () throws <exceptions>;		<res-type> get_XXX () raises (<exceptions>);
	void setXXX (<arg-list>) throws <exceptions>;		<res-type> set_XXX () raises (<exceptions>);

Table 5-4 Translation of EJB method requests into CCM operation requests

EJB Interface	Method called by client	CCM interface	Operation called by bridge
EJBMetadata	EJBHome getEJBHome () throws RemoteException;		Translation performed by bridge on all these invocations does not call a CCM standard operation.
	Class getHomeInterfaceClass () throws RemoteException;		
	Class getRemoteInterfaceClass () throws RemoteException;		
	Class getPrimaryKeyClass () throws RemoteException;		
	boolean isSession () throws RemoteException;		
	boolean isStatelessSession() throws RemoteException		

In addition, the EJB programming model allows a Java client to:

- Locate EJB homes and distinguished EJB objects via **JNDI**.
- Demarcate transactions via a **UserTransaction** object, after locating this object via **JNDI**.

These requests are translated into similar requests provided by the CORBA component programming model, as follows:

- Location of home and EJB objects requires the definition of a mapping of JNDI to the COSNaming service. It also requires the mapping of a COSNaming name space into a JNDI name space.
- Transaction demarcation requires the definition of a mapping of **JTA** to the CORBA transaction service. It also requires that a **JNDI** name space location be populated with an object that implements **UserTransaction** and that maps to the corresponding CORBA transaction service object.

5.4.3 Interoperability of the View

As stated in Section 5.4.2, “Translation of EJB requests into CORBA Component Requests,” on page 5-17 can happen at either the client-side, the server-side, or a combination of the two.

However, in order to provide interoperability of implementations of EJB views of CORBA components, a minimal number of translation points must be performed and they must be performed at an explicitly defined location: either the client-side or the server-side. For the implementation of an EJB view of a CORBA component, and for a CCM interface, the translation points are:

Translation of specific method names

The following methods shall translate their names as indicated.

Table 5-5 Translation of specific method names

EJB Interface	Method name	CCM Interface	Translation
EJBHome	remove	CCMHome	remove_component
<name>	findByPrimaryKey	<name>Implicit	find_by_primary_key
	remove__java_lang_Object		remove
	create__java_lang_Object		create
EJBObject	getEJBHome	CCMObject	get_ccm_home
	getPrimaryKey		get_primary_key
	isIdentical	CORBA::Object	is_equivalent

Handling of standard exceptions

The following exceptions, caught by the indicated methods, shall be translated as indicated before raising them to their EJB clients.

Table 5-6 Handling of standard exceptions

EJB Interface	Method name	Exception caught	Translation
EJBHome	remove	Components::RemoveFailure CORBA system exceptions	RemoveException RemoteException
	remove__java_lang_Object	Components::RemoveFailure CORBA system exceptions	RemoveException RemoteException
<name>	create	Components::CreateFailure Components::DuplicateKeyValue	CreateException DuplicateKeyException
	findByPrimaryKey	Components::UnknownKeyValue Components::FinderFailure	ObjectNotFoundException FinderException
EJBObject	getEJBHome	CORBA system exceptions	RemoteException
	getPrimaryKey	CORBA system exceptions	RemoteException
	remove	Components::RemoveFailure CORBA system exceptions	RemoveException RemoteException
	isIdentical	CORBA system exceptions	RemoteException

Note Ð CORBA system exceptions are translated into **RemoteException** according to rules defined in formal/01-06-07, section 1.4.8.

Handling of a primary key parameter

The methods **create** and **findByPrimaryKey**, defined by **<name>**, and **remove__java_lang_Object**, defined by **EJBHome**, shall translate the primary key valuetype they get as input parameter to a **CORBA::Any** equivalent.

The method **getPrimaryKey**, defined by **EJBObject**, shall translate the **CORBA::Any** value of the primary key it gets as a result from its request into an equivalent Java Object valuetype before returning it.

5.4.4 *Example*

We show a simple CORBA component definition and its corresponding EJB mapping. The basic CORBA component **Account** is defined in terms of a regular IDL interface **AccountOps**. The home **AccountHome** is defined to manage **Account** and to use a primary key.

```
interface AccountOps {
    void debit( in double amt ) raises (NotEnoughFunds);
    void credit( in double amt );
};

component Account supports AccountOps {
    readonly attribute double balance;
};

valuetype AccountKey {
    public long acctNo;
};

home AccountHome manages Account primarykey AccountKey {
    finder largeAccount( double threshold );
};
```

The following EJB definition is derived from the definition of **Account** and its home.

```
public interface Account extends javax.ejb.EJBObject {
    public void debit( double amount )
        throws NotEnoughFunds, java.rmi.RemoteException;
    public void credit( double amount ) throws java.rmi.RemoteException;
    public double getBalance() throws java.rmi.RemoteException;
};

public class AccountKey implements java.io.Serializable {
    public long acctNo;
    public AccountKey( long k ) { acctNo = k; }
};
```

```
public interface AccountHome extends javax.ejb.EJBHome {
    public Account create( AccountKey key )
        throws DuplicateKeyException, CreateException,
            java.rmi.RemoteException;
    public Account findByPrimaryKey( Account key )
        throws ObjectNotFoundException, FinderException,
                java.rmi.RemoteException;
    public Account findByLargeAccount( double threshold )
            throws java.rmi.RemoteException;
};
```

5.5 *Compliance with the Interoperability of Integration Views*

As stated in Section 5.3.3, ÒInteroperability of the View,Ó on page 5-10 and Section 5.4.3, ÒInteroperability of the View,Ó on page 5-19, request translations must happen at an explicitly defined location: either the client-side or the server side.

Rather than mandate one location arbitrarily, a number of levels of compliance with the interoperability of integration views are defined. Vendors shall clearly state what level of interoperability is supported by their implementations. These levels are:

- **NONE**: Integration view implementations that comply with this level actually perform no request translations. These implementations can still interoperate with other implementations that understand non-translated requests (e.g., implementations compliant with levels **SERVER-SIDE** and **FULL**).
- **CLIENT-SIDE**: Translation occurs either in the address space of a client stub or in a separate address space downstream from the client stub but before the resulting GIOP request gets sent to the server.
- **SERVER-SIDE**: Translation occurs either in the address space of a server skeleton or in a separate address space upstream from the server skeleton but after the GIOP request has been received from the client. The presence of a server-side view must not prevent native (i.e., non-translated) access to the component.
- **FULL**: Integration view implementations that comply with this level comply with both the **CLIENT-SIDE** and **SERVER-SIDE** levels. Note that a stand-alone bridge in a separate address space complies at this level since it is both upstream of the client (**SERVER-SIDE**) and downstream of the server (**CLIENT-SIDE**).
- **FULL**: Integration view implementations that comply with this level comply with both the **CLIENT** and the **SERVER** levels.

Table 5-7 illustrates the possible combinations of level compliance that are implied by the previous definitions. Rows in the table denote implementations compliant with a given level that send a request. Columns denote implementations compliant with a given level that receive a request. So, for example, a **SERVER-SIDE** implementation cannot interoperate with a **CLIENT-SIDE** implementation because the **SERVER-SIDE** implementation does not translate on send and the **CLIENT-SIDE** implementation does not translate on receive.

Table 5-7 Compliance with the Interoperability of Integration Views

	NONE	**CLIENT-SIDE**	**SERVER-SIDE**	**FULL**
NONE	no	no	yes	yes
CLIENT-SIDE	no	yes	yes	yes
SERVER-SIDE	no	no	yes	yes
FULL	yes	yes	yes	yes

5.6 Comparing CCM and EJB

The following series of tables summarized the component APIs for Enterprise Java Beans (EJB 1.1) and Basic CORBA Components. The tables are organized as follows:

1. The home interfaces that define the remote access protocols for creating or finding EJBs or CORBA components (Section 5.6.1, ÒThe Home Interfaces,Ó on page 5-23).
2. The component interfaces that define the remote access protocols for invoking business operations on EJBs or CORBA components (Section 5.6.2, ÒThe Component Interfaces,Ó on page 5-25).
3. The callback interfaces that the CORBA component or EJB programmer must implement (Section 5.6.3, ÒThe Callback Interfaces,Ó on page 5-26).
4. The Context interfaces that provide the component developer access to container-provided services (Section 5.6.4, ÒThe Context Interfaces,Ó on page 5-28).
5. The Transaction interface that supports bean-managed or component-managed transactions (Section 5.6.5, ÒThe Transaction Interfaces,Ó on page 5-29).
6. The metadata interfaces that support access to component metadata (Section 5.6.6, ÒThe Metadata Interfaces,Ó on page 5-30).

5.6.1 The Home Interfaces

Table 5-8 compares the home interfaces and operations that make up the EJB and CORBA component models. In EJB, the **EJBHome** object is created by the EJB container providerÕs tools and provides implementations for methods of the base class and delegates factory or finder methods on a derived class (**<name>Home**) to similarly named methods on the bean itself (**<name>Bean**).

In the CORBA component model, homes are defined as righteous CORBA objects and the associated factory or finder methods are generated as operations on the home and the component developer implements these directly so the container need not provide delegation support. The component developer may not even need to provide implementations for the default factory and finder operations if sufficient information is provided with the componentÕs definition.

For CORBA clients to use EJB implementations, the container provider must externalize **EJBHome** to the CORBA client as a CORBA component home. This is accomplished by extensions to the Java to IDL mapping defined in Chapter 8. For EJB clients to access CORBA component homes, the container provider must create an **EJBHome** object that serves as a bridge between equivalent operations on **EJBHome** and the CORBA component home. This bridge is also described in Chapter 8.

Table 5-8 Comparing the home interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components	
Interface	EJBHome extends java.rmi.Remote	CCMHome	
Operation	public EJBMetaData get EJBMetaData () throws java.rmi.RemoteException	ComponentDef get_component_def ();	CORBA IR supports more metadata
	public HomeHandle getHomeHandle() throws java.rmi.RemoteException		CORBA::object_to_string provides same function
	public void remove (HomeHandle handle) throws java.rmi.RemoteException, RemoveException	void remove_component (in CCMObject component) raises (CCMException);	CORBA references instead of handles REMOVE_ERROR is minor code
	public void remove (java.lang.Object primaryKey) throws java.rmi.RemoteException, RemoveException		similar operation is defined on <home>Implicit for Homes with primarykey
Interface	HomeHandle extends java.io.Serializable		CORBA reference used for handle
	public EJBHome getEJBHome() throws java.rmi.RemoteException		CORBA::string_to_object
Module	<session-name>	<session-home>	
Interface	<session>home extends EJBHome	<session-home>::CCMHome, <session-home>Implicit, <session-home>Explicit	
Operation	public <session-name>Remote create (<arg-type> <arg-list>) throws CreateException	<session-component> create ();	Generated operation Inherited from <home>Implicit
Module	<entity-name>	<entity-home>	
Interface	<entity>home extends EJBHome	<entity-home>::CCMHome, <entity-home>Implicit, <entity-home>Explicit	

Table 5-8 Comparing the home interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Operation	public <entity-name>Remote create (<arg-type> <arg-list>) throws CreateException, DuplicateKeyException	<entity-component> create () raises (InvalidKey, DuplicateKey);	Generated operation Inherited from <home>Implicit
	public <entity-name>Remote findByPrimaryKey (<arg-type> <arg-list>) throws FinderException, ObjectNotFoundException	<entity-component> find (in <key-type> key) raises (InvalidKey, UnknownKeyType);	Generated operation Inherited from <home>Implicit
	public <entity-name>Remote find<method> (<arg-type> <arg-list>) throws FinderException, ObjectNotFoundException	<entity-component> <find-method> (in <arg-type> <arg-list>) raises (<exceptions>);	Specified operation Inherited from <home>Explicit

5.6.2 The Component Interfaces

Table 5-9 compares the component interfaces and operations that make up the EJB and CORBA component models. In EJB, the **EJBObject** object is created by the EJB container providerÕs tools and provides implementations for methods of the base class and delegates business methods to a derived class (**<name>Remote**).

In the basic CORBA component model, components are defined as righteous CORBA objects and the associated business methods are defined as operations on a supported interface and the component developer implements these directly so the container need not provided delegation support.

For CORBA clients to use EJB implementations, the container provider must externalize **EJBObject** to the CORBA client as a CORBA component. This is accomplished by extensions to the Java to IDL mapping defined in Chapter 8. For EJB clients to access CORBA components, the container provider must create an **EJBObject** implementation that serves as a bridge between business methods on **EJBObject** and the basic CORBA componentÕs supported interface. This bridge is also described in Chapter 8.

Table 5-9 Comparing the remote interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components	
Interface	EJBObject extends java.rmi.Remote	CCMObject	
Operation	public EJBHome getEJBHome() throws java.rmi.RemoteException	CCMHome get_ccm_home();	
	public java.lang.Object primaryKey getPrimaryKey() throws java.rmi.RemoteException		operation defined on <entity>home

Table 5-9 Comparing the remote interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Operation (continued)	public void remove (Handle handle) throws java.rmi.RemoteException, RemoveException	void remove() raises (CCMException);	CORBA references instead of handles; REMOVE_ERROR is minor code
	public Handle getHandle() throws java.rmi.RemoteException		CORBA::object_to_string
	public boolean isIdentical (EJBObject obj) throws java.rmi.RemoteException	boolean is_equivalent(in Object obj);	
Interface	Handle extends java.io.Serializable		CORBA reference used for handle
	public EJBObject getEJBObject() throws java.rmi.RemoteException		CORBA::string_to_object
Module	<session-bean>	<session-component>	
Interface	<session>Remote extends EJBObject	<session>::CCMObject	
	<res-type> <operation> (<arg-type> <arg-list>) throws <exceptions>	<res-type> <operation>(in <arg-type> <arg-list) raises (<exceptions>);	business methods
Module	<entity-bean>	<entity-component>	
Interface	<entity>Remote extends EJBObject	<entity>::CCMObject	
	<res-type> <operation> (<arg-type> <arg-list>) throws <exceptions>	<res-type> <operation> (in <arg-type> <arg-list) raises (<exceptions>);	business methods

5.6.3 The Callback Interfaces

Table 5-10 summarizes the callback interfaces the EJB programmer or basic CORBA component programmer must implement. The EJB interfaces are specified as Java interfaces in accordance with the EJB 1.1 specification dated June 28, 1999. The CCM interfaces are specified in IDL as defined in this specification.

Table 5-10 Comparing EJB and CCM Callback Interfaces

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components::Basic	
Interface	EnterpriseBean	EnterpriseComponent	
Interface	SessionBean extends EnterpriseBean	SessionComponent::EnterpriseComponent	
Operation	public void setSessionContext (SessionContext ctx) throws EJBException	void set_session_context(in SessionContext ctx) raises (CCMException);	
	public void ejbActivate () throws EJBException	void ccm_activate () raises (CCMException);	

Table 5-10 Comparing EJB and CCM Callback Interfaces

Construct	EJB Form	CCM Form	Notes
Operation (continued)	public void ejbPassivate () throws EJBException	void ccm_passivate () raises (CCMException);	
	public void ejbRemove () throws EJBException	void ccm_remove () raises (CCMException);	
Interface	<name>Bean extends SessionBean		Home operations are not delegated in CCM.
Operation	public void ejbCreate (<Arg-type> <arg-list>) throws CreateException, EJBException)		Implemented on home, CREATE_ERROR is minor code
Interface	SessionSynchronization	SessionSynchronization	
Operation	public void afterBegin () throws EJBException	void after_begin () raises (CCMException);	
	public void beforeCompletion() throws EJBException	void before_completion () raises (CCMException);	
	public void afterCompletion (boolean committed) throws EJBException	void after_completion (in boolean committed) raises (CCMException);	
Interface	EntityBean extends EnterpriseBean	EntityComponent::EnterpriseComponent	
Operation	public void setEntityContext (EntityContext ctx) throws EJBException	void set_entity_context (in EntityContext ctx) raises CCMException;	
	public void unsetEntityContext () throws EJBException	void unset_entity_context () raises (CCMException);	
	public void ejbActivate () throws EJBException	void ccm_activate () raises (CCMException);	
	public void ejbLoad () throws EJBException	void ccm_load () raises (CCMException);	
	public void ejbStore () throws EJBException	void ccm_store() raises (CCMException);	
	public void ejbPassivate () throws EJBException	void ccm_passivate () raises (CCMException);	
	public void ejbRemove () throws RemoveException, EJBException	void ccm_remove () raises (CCMException);	REMOVE_ERROR is a minor code
Interface	<name>Bean extends EntityBean		Home operations are not delegated in CCM.
Operation	public <key-type> ejbcreate (<Arg-type> <arg-list>) throws CreateException, DuplicateKeyException, EJBException		Implemented on home, CREATE_ERROR and DUPLICATE_KEY are minor codes

Table 5-10 Comparing EJB and CCM Callback Interfaces

Construct	EJB Form	CCM Form	Notes
	public void ejbPostCreate () throws CreateException, DuplicateKeyException, EJBException		post_create not required in CCM due to CORBA identity model
	public <key-type> findByPrimaryKey (<Arg-type> <arg-list>) throws FinderException, NoSuchEntityException, ObjectNotFoundException, EJBException		Implemented on home, FIND_ERROR, NO_SUCH_ENTITY and OBJECT_NOT_FOUND are minor codes
	public <key-type> find<method> (<Arg-type> <arg-list>) throws FinderException, NoSuchEntityException, ObjectNotFoundException, EJBException		Implemented on home, FIND_ERROR, NO_SUCH_ENTITY and OBJECT_NOT_FOUND are minor codes

5.6.4 The Context Interfaces

The context interfaces summarized in Table 5-11 provide accessors to services provided by the component container. They are used by the component developer when these services are required.

Table 5-11 Comparing the EJB and CCM Context Interfaces

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components::Basic	
Interface	EJBContext	CCMContext	
Operation	public java.security.Principal getCallerPrincipal()	Principal get_caller_principal();	
	public EJBHome getEJBHome()	CCMHome get_ccm_home();	
	public boolean getRollbackOnly() throws java.lang.IllegalState	boolean get_rollback_only() raises (IllegalState);	
	public javax.transaction.UserTransaction getUserTransaction () throws java.lang.IllegalState	Transaction::UserTransaction get_user_transaction () raises (IllegalState);	
	public boolean isCallerInRole (java.lang.String (roleName)	boolean is_caller_in_role (in string role);	
	public void setRollbackOnly() throws java.lang.IllegalState	void set_rollback_only() raises IllegalState;	
Interface	SessionContext extends EJBContext	SessionContext::CCMContext	
Operation	public EJBObject getEJBObject() throws java.lang.IllegalState	CORBA::Object get_CCM_Object() raises (IllegalState);	this will be the component reference

Table 5-11 Comparing the EJB and CCM Context Interfaces

Construct	EJB Form	CCM Form	Notes
Interface	EntityContext extends EJBContext	EntityContext::CCMContext	
Operation	public EJBObject getEJBObject() throws java.lang.IllegalState	CORBA::Object get_CCM_Object() raises (IllegalState);	this will be the component reference
	public java.lang.Object getPrimaryKey () throws java.lang.IllegalState	PrimaryKeyBase get_primary_key() raises (IllegalState);	

5.6.5 The Transaction Interfaces

Table 5-12 summarizes the transaction interfaces provided for bean-managed or component-managed transactions. Both EJB and CCM provide an accessor function in the context to obtain a reference to a transaction service. The transaction service supported for EJB is JTA, a subset of JTS which is equivalent to the CORBA transaction service (OTS). The transaction service supported for CORBA components is implemented by the component container as a wrapper over the CORBA transaction service. **Components::Transaction** is functionally equivalent to JTA (which is not a distinct compliance level for OTS) with the addition of **suspend** and **resume**.

Table 5-12 Comparing the EJB Transaction service (JTA) with CORBA component transactions

Construct	EJB Form	CCM Form	Notes
Module	javax.transaction	Components::Transaction	
Interface	UserTransaction	UserTransaction	
Operation	public void begin() throws NotSupported, SystemException	void begin () raises (NotSupported, SystemError);	SystemError to avoid confusion with System Exception
	public void commit() throws RollbackException, HeuristicMixedException, HeuristicRollbackException, java.security.SecurityException, java.lang.IllegalStateException, SystemException	void commit() raises (RollbackError, HeuristicMixed, HeuristicRollback, Security, IllegalState, SystemError	map CORBA system exceptions TRANSACTION_ROLLED_BACK to ROLLBACK and NO_IMPLEMENT to SECURITY
	public void rollback() throws java.security.SecurityException, java.lang.IllegalStateException, SystemException	void rollback() raises (Security, IllegalState, SystemError);	
	public void setRollbackOnly() throws SystemException	void set_rollback_only() raises (SystemError);	
	public int getStatus() throws SystemException;	Status get_status() raises (SystemError);	

Table 5-12 Comparing the EJB Transaction service (JTA) with CORBA component transactions

Construct	EJB Form	CCM Form	Notes
	public void setTransactionTimeout (int seconds) throws SystemException	void set_transaction_timeout(in long to) raises (SystemError);	
		TranToken suspend() raises (NoTransaction, SystemError);	CCM supports suspend/resume which JTA does not
		void resume(in TranToken) raises (invalidToken, SystemError);	CCM supports suspend/resume which JTA does not

5.6.6 *The Metadata Interfaces*

The EJB component model supports a limited set of metadata through the **EJBMetaData** interface. The CORBA component model extends the CORBA interface repository to add component-unique metadata for components. This metadata is in addition to the metadata currently provided by the IR. When EJB clients access CORBA components, the container provider must provide an implementation of **EJBMetaData**, which supports the necessary metadata from the Interface Repository or the component descriptors. This is described further in Chapter 8. When CORBA clients access EJB implementations, the Interface Repository is already populated for the **EJBHome** and **EJBObject** interfaces, enabling client requests to be satisfied. Table 5-13 compares the metadata supported by EJB and CORBA Components.

Table 5-13 Comparing component metadata between EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	IR	
Interface	EJBMetaData	ComponentDef	
	public EJBHome getEJBHome()		
	public java.lang.Class getHomeInterfaceClass()		
	public java.lang.Class getRemoteInterfaceClass()		
	public java.lang.Class getPrimaryKeyClass()		
	public boolean isSession()		
	public boolean isStatelessSession()		

Packaging and Deployment 6

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter describes the CORBA component packaging and deployment model.

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒIntroductionÓ	6-1
ÒComponent PackagingÓ	6-2
ÒSoftware Package DescriptorÓ	6-3
ÒCORBA Component DescriptorÓ	6-16
ÒComponent Assembly PackagingÓ	6-40
ÒComponent Assembly FileÓ	6-40
ÒComponent Assembly DescriptorÓ	6-41
ÒProperty File DescriptorÓ	6-62
ÒComponent DeploymentÓ	6-68

6.1 Introduction

Component implementations may be packaged and deployed.

A CORBA Component package maintains one or more implementations of a 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 one or more descriptors and a set of files. The descriptors describe the characteristics of the package and point 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 specialization of a general software package. The software packaging scheme, described here, could be used to package arbitrary software entities. In fact it was initially inspired by the Open Software Description (OSD) note to the W3C. 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 alone, as is, or it may be included in a component assembly package and deployed as part of the assembly along with the other components 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 interface ports, represented by *provides* and *uses* features and between event ports, represented by *emits*, *produces*, and *consumes* features.

Component and assembly packages are provided as input to a deployment tool.

A deployment tool deploys individual components and assemblies of components to an installation site, usually a set of hosts on a network. The user of the deployment tool guides in determining where each component should be installed. Components within an assembly may be installed on a single machine or scattered across a network.

Based on an assembly descriptor and user input, the deployment tool installs and activates component homes and instances; it configures component properties and connects components together via interface and event ports, as indicated in the assembly descriptor.

6.2 Component Packaging

A software package is represented by a descriptor and a set of files. The descriptor and associated files are grouped together in a ZIP archive file. The software package could be used to describe arbitrary software packages.

In relation to CORBA Components, software packages are used to package a CORBA Component implementation.

6.3 Software Package Descriptor

The contents of a software package 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 is used to describe component software packages. The descriptor file has a Ò.csdÓ 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 Òmeta-inf.Ó

The structure and intent of the descriptor can be better understood by looking at an example.

6.3.1 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" />
```

Note Ð Issue 5091

```
     <idl id="IDL:M1/Bank:1.0" homeid="IDL:M1/BankHome:1.0">
     <link href=Óftp://x/y/Bank.idlÓ/></idl>

     <propertyfile><fileinarchive name="bankprops.cpf"/></propertyfile>

     <implementation id=ÓDCE:700dc518-0110-11ce-ac8f-0800090b5d3eÓ>
          <os name="WinNT" version="4,0,0,0" />
          <os name="Win95" />
          <processor name="x86" />
          <compiler name="MyFavoriteCompiler" />
          <programminglanguage name="C++" />

          <dependency type=ÓORBÓ>
                    <name>ExORB</name>
         </dependency>

          <descriptor type=ÓCORBAComponentÓ>
                    <fileinarchive name="processcontainer.ccd" />
         </descriptor>

         <code type=ÓDLLÓ>
    <fileinarchive name="bank.dll"/>
<entrypoint>createBankHome</entrypoint>
         </code>
```

```
        <dependency type=ÓDLLÓ>
    <localfile name="rwthr.dll"/>
        </dependency>

    </implementation>

    <implementation id=ÓDCE:297f3e18-0110-11ce-ac8f-08074982ad3eÓ
                                    variation=ÓRemoteHomeÓ>
        <os name="Solaris" version="5,5,0,0" />
        <processor name="sparc" />
        <!-- . . . -->
    </implementation>

    <implementation> <!-- another implementation --> </implementation>
</softpkg>
```

6.3.2 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 softpkg.dtd in Section 7.1, Òsoftpkg.dtd,Ó on page 7-1.

Note Ð An effective strategy for studying an XML DTD is to recursively navigate from the root element, which in this case is **softpkg**, to each child element.

6.3.2.1 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 allows the component implementor to provide specialized 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
)* >
```

Note Ð Issue 5584

```
<!ATTLIST softpkg
        name        ID       #REQUIRED
        version     CDATA #~~OPTIONAL~~IMPLIED >
```

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., Ò1,0,0,0Ó).

6.3.2.2 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
       )* >
```

6.3.2.3 The code Element

The **code** element points to a file in the archive that 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 to the code. The optional **usage** element is used to describe how to use (i.e., invoke, the code).

```
<!ELEMENT code
            (     ( codebase
                  | fileinarchive
                  | link
                  )
            , entrypoint?
            , usage?
            ) >
```

```
<!ATTLIST code
        type CDATA #IMPLIED >
```

The **type** attribute specifies the type of code. The types Ò**DLL**,Ó Ò**Executable**,Ó and Ò**Java Class**Ó shall be recognized as valid types.

6.3.2.4 The codebase Element

The **codebase** element is used to specify a resource. If the resource isnÕt 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.

6.3.2.5 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 ) >
```

6.3.2.6 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 Òw,x,y,zÓ format.

6.3.2.7 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
       ( softpkgref
       | codebase
       | fileinarchive
       | localfile
       | name
       | valuetypefactory
       ) >
<!ATTLIST dependency
       type   CDATA              #IMPLIED
       action (assert | install)"assert">
```

The **type** attribute specifies the type of the resource required. The types Ò**DLL**,Ó Ò**Executable**,Ó and Ò**Java Class**Ó shall be recognized as valid types.

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.

6.3.2.8 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 ) >
```

6.3.2.9 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 CORBA Component descriptor.

```
<!ELEMENT descriptor
            ( link
            | fileinarchive
            ) >
<!ATTLIST descriptor
            type   CDATA  #IMPLIED>
```

The **type** attribute is the type of the descriptor.

Note Ð With respect to the CORBA Component model, a **type** of ÒCORBA ComponentÓ is used to indicate a CORBA component descriptor (described in Section 6.4.4, ÒCORBA Component Descriptor Example,Ó on page 6-18).

6.3.2.10 The entrypoint Element

The entrypoint element specifies the entry point to a software package. See Section 6.9.8, ÒComponent Entry Points (Component Home Factories),Ó on page 6-81 for information on CORBA component entry points.

```
<!ELEMENT entrypoint ( #PCDATA ) >
```

6.3.2.11 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 **PCDATA**, meaning that it can have character data or markup.

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 (#PCDATA) >
<!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 elementÕs **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 that 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., Ò<em>Ó).

6.3.2.12 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.

6.3.2.13 The humanlanguage Element

The **humanlanguage** element specifies a spoken language. **humanlanguage** has an EMPTY content model.

```
<!ELEMENT humanlanguage EMPTY >
<!ATTLIST humanlanguage
          name CDATA #REQUIRED >
```

The human language name is specified in the **name** attribute.

6.3.2.14 The idl Element

Note Ð Issue 5091

The **idl** element points to a file or repository containing ~~an idl~~ IDL definitions. If a component and its home are defined in different files or repositories, then the **idl** element refers to the file or repository where the home is defined.

```
<!ELEMENT idl
          ( link
          | fileinarchive
          | repository
          ) >
<!ATTLIST idl
          id CDATA #REQUIRED
          homeid CDATA #REQUIRED >
```

The **id** attribute is a repository Id that uniquely identifies the IDL equivalent interface for the software component. The **homeid** attribute is a repository Id that uniquely identifies the IDL equivalent interface for the home of the software component.

6.3.2.15 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, descriptors, dependencies, code filename, entry points, and other characteristics.

```
<!ELEMENT implementation
        ( description
        | code
        | compiler
        | dependency
        | descriptor
        | extension
        | programminglanguage
        | humanlanguage
        | os
        | propertyfile
        | processor
        | runtime
        )* >
<!ATTLIST implementation
        id ID #IMPLIED
        variation CDATA #IMPLIED >
```

The **id** attribute is a DCE UUID that uniquely identifies the implementation.

The **variation** attribute is used to indicate a variation from a normal implementation. The interpretation of the variation attribute depends on user of the softpkg.

Note Ð The only valid variation string defined by the CORBA Component model is Ò**ProxyHome**.Ó The ProxyHome variation indicates that the component implementation contains a proxy home only, not a full component implementation.

6.3.2.16 The implref Element

The **implref** element is used to refer to an implementation within a softpkg.

```
<!ELEMENT implref EMPTY >
<!ATTLIST implref
        idref CDATA #REQUIRED >
```

The **idref** attribute refers to a unique **implementation** element **id** in the softpkg descriptor.

6.3.2.17 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; >
```

6.3.2.18 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; >
```

6.3.2.19 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.

6.3.2.20 The name Element

The **name** element is an optional child element of both the **author** and **dependency** elements. When used as a child of **author**, it specifies the name of the author. When used as a child of **dependency**, it specifies the expected value of the dependency. It has string content.

```
<!ELEMENT name ( #PCDATA ) >
```

6.3.2.21 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 Òw,x,y,zÓ 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

6.3.2.22 *The pkgtype Element*

The **pkgtype** element is used to identify the type of software that the **softpkg** represents. This specification reserves package types Ò**CORBA Component**Ó and Ò**CORBA Interface Impl**Ó for the packaging of CORBA component and interface implementations.

```
<!ELEMENT pkgtype ( #PCDATA ) >
<!ATTLIST pkgtype
    version CDATA #IMPLIED >
```

The optional **version** attribute specifies a version of the package type.

6.3.2.23 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/390

6.3.2.24 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.

6.3.2.25 The propertyfile Element

The **propertyfile** element is used to refer to a property file associated with the **softpkg** or implementation.

A property file of a particular type, defined at the top level of the descriptor, may be overridden by implementation specific property files of that type, defined in an implementation element.

```
<!ELEMENT propertyfile
```

```
             ( fileinarchive
             | link) >
<!ATTLIST propertyfile
             type CDATA #IMPLIED >
```

The **type** attribute, distinguishes a property file from other types of property files. If there is only one type of property file, or if the type of property file is implicit given a context, then the **type** is not required.

6.3.2.26 The repository Element

The **repository** element is a child element of the **idl** element used to point to a repository, such as the interface repository.

```
<!ELEMENT repository
        ( ins
        | objref
        | link
        ) >
<!ATTLIST repository
        type CDATA #IMPLIED >
```

The **type** attribute specifies the type of repository. Currently, the only predefined value for **type** is Ò**CORBA Interface Repository**.Ó

6.3.2.27 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 Òw,x,y,zÓ format.

6.3.2.28 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.

Note Ð In the context of CORBA Components, the **href** attribute may be used to specify INS format names.

To demonstrate the usage of an element that employs the **simple-link-attributes** entity, consider the following element definition:

```
<!ELEMENT          exampleelement EMPTY >
<!ATTLIST          exampleelement
          %simple-link-attributes; >
```

This could be used as follows:

```
<exampleelement href=Òhttp://www.abc.com/xyzÓ />
```

6.3.2.29 The softpkg Element

This is the root element of the descriptor. See Section 6.3.2.1, ÒThe softpkg Root Element,Ó on page 6-4.

6.3.2.30 The softpkgref Element

The **softpkgref** element refers to an external softpkg. The file is referenced by a **fileinarchive** element or a **link**. An optional **implref** element refers to a particular implementation within the softpkg descriptor.

```
<!ELEMENT softpkgref
          (    ( fileinarchive
               | link
               )
               , implref?
               ) >
```

6.3.2.31 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 ) >
```

6.3.2.32 The usage Element

The **usage** element contains a string usage description.

```
<!ELEMENT usage ( #PCDATA ) >
```

6.3.2.33 The valuetypefactory Element

The **valuetypefactory** element contains information needed to register and utilize a valuetype factory. The **valuetypefactory** element is a child element of the **dependency** element.

```
<!ELEMENT valuetypefactory
    ( codebase
    | fileinarchive
    | link
    ) >
<!ATTLIST valuetypefactory
    repid               CDATA #REQUIRED
    valueentrypoint     CDATA #IMPLIED
    factoryentrypoint CDATA #IMPLIED >
```

The **repid** attribute specifies the repository id of the valuetype created by the valuetype factory. The **factoryentrypoint** attribute specifies an operation or function that can be used to create an instance of a valuetype factory associated to the repository id given by **repid**. The **valueentrypoint** attribute specifies an operation or function that can be used to create an instance of a valuetype using a factory instance created with **factoryentrypoint**.

6.3.2.34 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; >
```

6.4 CORBA Component Descriptor

The CORBA Component descriptor describes a component. It is referred to by a **<descriptor type=”CORBAComponent>** element in a softpkg descriptor describing a CORBA component.The CORBA Component descriptor specifies component characteristics, used at design and deployment time. A component descriptor file has a recommended Ò.ccdÓ extension, standing for CORBA Component Descriptor.

The component descriptor is generated by a CIDL compiler. This is convenient as the CIDL compiler has much of the necessary information at hand. However, the compiler doesnÕt have all of the information required. The user, likely with the help of a packaging tool, 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 packaging tool.

The component descriptor is described using an XML vocabulary. The complete XML DTD for the descriptor is in the *XML DTDs* chapter. This section discusses each element of the descriptor in detail.

6.4.1 Component Feature Description

The component descriptor provides information that a design tool may use to display information about a component. This includes information about the interfaces that the component supports and its ports.

Note Ð For the purpose of component packaging and deployment we will use the term *ports* to collectively describe the interfaces that a component uses and provides and the events that it emits, publishes, and consumes. In addition, provides and uses ports will be called *interface ports*, and emits, publishes, and consumes ports will be termed *event ports*.

The component descriptor describes the structure of a component with respect to supported interfaces, inherited components, and uses and provides ports. The component is described by a **componentfeatures** element, which describes inherited components, supported interfaces, used and provided interfaces, and emitted, published and consumed events. If the component inherits from other components, then the features of that component are described in a separate **componentfeatures** element and referenced by the **inheritscomponent**. The primary **componentfeatures** element of the descriptor is indicated by the **repositoryid** element of the component descriptor.

Each interface supported or provided by a component is described by an **interface** element. Interface elements are referenced by the repository id of the interface. An **interface** has a name and a repository id, and may inherit from other interfaces. The inheritance relationship is represented by the **inheritsinterface** element.

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 that *uses* interface *X* could be *connected* to another component that *provides* interface *X*, based on information in each componentÕs descriptor.

6.4.2 Deployment Information

At deployment time, the component descriptor is used to determine the type of container in which the component needs to be installed and to provide information about the component to the container.

The **componentkind** element tells the creator of the container what kind of container to create. A **componentkind** can be either **session**, **service**, **process**, or **entity**.

¥ The **transaction** element indicates the transactional characteristic of the component.

¥ The **eventpolicy** is used to indicate the quality of service of event ports.

¥ The **threading** element indicates how the container should dispatch operations on the component instance. If **threading** is set to *multithread*, then the component is ready to accept multiple threads of control within a single instance. The component takes responsibility for protecting its internal state. If **threading** is set to *serialize*, then the container will serialize all calls to a single instance. Note that although the component will not need to protect instance state, the container may employ other threads to invoke other instances of the component type, thus the component must protect any static or class data.

The **configurationcomplete** element tells the deployment agent whether the component expects for **configuration_complete** to be called after its properties have been set and its ports configured to their initial state (e.g., as described by a component assembly descriptor).

The **segments** element provides the container with information necessary to map segment tags to segment names, segment tags to facet tags, and segment tags to abstract storage home types. The **facettag** element references a **provides** interface element described elsewhere in the descriptor. The **provides** element maps facet tags to provided interface names. A container uses the information provided by these elements to construct data structures mapping segment tags to segment names, facet tags to facet names, and segment tags to facet tags. Note that a segment tag can map to more than 1 facet tag.

6.4.3 CIDL Compiler Responsibilities

A CIDL compiler is responsible for generating an initial component descriptor. This initial descriptor is vendor specific and may be manipulated directly by the user or using vendor supplied tools.

6.4.4 CORBA Component Descriptor Example

```
<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "corbacomponent.dtd">

<corbacomponent>
   <corbaversion> 3.0 </corbaversion>
   <componentrepid repid="IDL:BookStore:1.0" />
   <homerepid repid="IDL:BookStoreHome:1.0" />
   <componentkind>
      <entity>
```

Note Ð Issue 5492

```
         <servant lifetime="~~process~~container" />
      </entity>
   </componentkind>
   <security rightsfamily="corba" />
   <threading policy="multithread" />
   <configurationcomplete set="true" />

   <segment name="bookseg" segmenttag="1">
      <segmentmember facettag="1" />
      <segmentmember facettag="2" />
      <containermanagedpersistence>
         <storagehome id="PSDL:BookHome:1.0" />
```

```
    <pssimplementation id="ACME-PSS" />
```

Note Ð Issue 5499

```
    ~~<catalog type="PSDL:BookCatalog:1.0" />~~
    <accessmode mode="READ_ONLY" />
    <psstransaction policy="TRANSACTIONAL" >
      <psstransactionisolationlevel level="SERIALIZABLE" />
    </psstransaction>
    <params>
      <param name="x" value="1" />
    </params>
  </containermanagedpersistence>
</segment>

<homefeatures                          name="BookStoreHome"
                                       repid="IDL:BookStoreHome:1.0">
  <operationpolicies>
    <operation name="*">
      <transaction use="never" />
    </operation>
  </operationpolicies>
</homefeatures>

<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"
      facettag="1">
      <operationpolicies>
        <operation name="getByAuthor">
          <requiredrights>
            <right name="get"/>
          </requiredrights>
        </operation>
        <operation name="getByTitle">
          <requiredrights>
            <right name="get"/>
          </requiredrights>
        </operation>
        <operation name="getByISBN">
          <requiredrights>
            <right name="get"/>
          </requiredrights>
        </operation>
      </operationpolicies>
    </provides>
    <provides
      providesname="shopping_cart"
```

```
         repid="IDL:CartFactory:1.0"
         facettag="2" />
      <uses
         usesname="ups_rates"
         repid="IDL:ShippingRates:1.0" />
      <uses
         usesname="fedex_rates"
         repid="IDL:ShippingRates:1.0" />
      <emits
         emitsname="low_stock"
         eventtype="StockRecord">
         <eventpolicy policy="normal" />
      </emits>
      <publishes
         publishesname="offer_alert"
         eventtype="SpecialOffer">
         <eventpolicy policy="normal" />
      </publishes>
   </ports>
</componentfeatures>

<componentfeatures name="Store" repid="IDL:Acme/Store">
   <supportsinterface repid="IDL:Acme/GeneralStore">
      <operationpolicies>
         <operation name="*">
            <transaction use="required" />
         </operation>
      </operationpolicies>
   </supportsinterface>
   <ports>
      <provides
         providesname="admin"
         repid="IDL:Acme/StoreAdmin:1.0"
         facettag="3" />
   </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">
   <operationpolicies>
      <operation name="*">
         <transaction use="required" />
         <requiredrights>
            <right name="manage"/>
            <right name="set"/>
         </requiredrights>
```

```
            </operation>
        </operationpolicies>
    </interface>
    <interface name="GeneralStore" repid="IDL:Acme/GeneralStore:1.0"/>

</corbacomponent>
```

6.4.5 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 Section 7.2, Òcorbacomponent.dtd,Ó on page 7-5.

6.4.5.1 The corbacomponent Root Element

The **corbacomponent** element is the root element of the CORBA component descriptor.

```
<!ELEMENT corbacomponent
          ( corbaversion
          , componentrepid
          , homerepid
          , componentkind
          , interop?
          , transaction?
          , security?
          , threading
          , configurationcomplete
          , extendedpoapolicy*
          , repository?
          , segment*
          , componentproperties?
          , homeproperties?
          , homefeatures+
          , componentfeatures+
          , interface*
          , extension*
          ) >
```

These elements must be provided in the order presented.

¥ **corbaversion** tells which version of CORBA the component is assuming.

¥ **componentrepid** is the interface repository id of the component. It also refers to a **componentfeatures** element later in the descriptor.

¥ **homerepid** is the interface repository id of the home. It also refers to a **homefeatures** element later in the descriptor.

- **componentkind** describes properties of the component that will determine what kind of container the component must reside in.
- **interop** specifies interoperation information (e.g., with EJB).
- **transaction** determines transaction policies for the entire component. This policy is optional and may be overridden on individual facets or supported interfaces.
- **security** specifies CORBA security rights family for the component.
- **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 after all of its properties have been set and its ports have been connected.
- **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.
- **segment** describes a segment including its name, tag, member facets, and storage home type.
- homefeatures describes the structure of the componentÕs homes.
- componentproperties specifies the default component properties file.
- homeproperties specifies the default home properties file.
- **componentfeatures** describes inherited components, supported interfaces, uses and provides ports, and emits, publish, 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.

See Section 7.2, Òcorbacomponent.dtd,Ó on page 7-5 for the full text of the component descriptor DTD.

6.4.5.2 The accessmode Element

Child element of **containermanagedpersistence**.

The **accessmode** element identifies whether the persistent state may be read and written or only read.

```
<!ELEMENT accessmode EMPTY>
<!ATTLIST accessmode
        mode (READ_ONLY|READ_WRITE) #REQUIRED >
```

The **mode** attribute identifies the access mode.

Note Ð Issue 5499

~~6.4.5.3 The catalog Element~~

~~Child element of **containermanagedpersistence**.~~

~~The **catalog** element identifies the catalog to be used in loading and storing persistent state.~~

~~<!ELEMENT catalog EMPTY>~~
~~<!ATTLIST catalog~~
~~type CDATA #REQUIRED >~~

~~The **type** attribute identifies the type of catalog.~~

6.4.5.4 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, publish, and consumes ports. A component also has the features that it inherits from other components. In addition, supported interfaces may inherit from other interfaces. By following the inheritance chain, a graph is formed from the primary component to a set of ports, supported interfaces, and other components. The root component in this graph is identified by the **repositoryid** child element of **corbacomponent**.

Note Ð Issue 5493

The information obtained by traversing the componentfeatures graph may be displayed by graphical tools. But more importantly, it allows component assembly tools to decide ~~what~~which ports on a component are capable of connecting to ports on other components.

```
<!ELEMENT componentfeatures
        ( inheritscomponent?
         , supportsinterface*
```

```
             , ports
             , operationpolicies?
             , extension*
             ) >
<!ATTLIST componentfeatures
             name  CDATA #REQUIRED
             repid CDATA #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).

6.4.5.5 *The componentkind Element*

Child element of **corbacomponent**.

The **componentkind** element defines the component category. For more information on these categories, see Section 4.1.4, ÒComponent Categories,Ó on page 4-5.

```
<!ELEMENT componentkind
     ( service
     | session
     | process
     | entity
     | unclassified
     ) >
```

6.4.5.6 *The componentproperties Element*

The **componentproperties** element specifies a default component property file. The format of the property file is described in Section 6.8, ÒProperty File Descriptor,Ó on page 6-62.

```
<!ELEMENT componentproperties
                                           ( fileinarchive
                                            ) >
```

6.4.5.7 *The componentrepid Element*

Child element of **corbacomponent**.

componentrepid 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 componentrepid EMPTY >
<!ATTLIST componentrepid
```

```
          repid CDATA #IMPLIED >
```

repid is the fully qualified repository id of the component.

6.4.5.8 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 >
```

6.4.5.9 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. The **eventpolicy** allows the transaction policy of the event port to be specified.

```
<!ELEMENT consumes
           ( eventpolicy
          , extension* ) >
<!ATTLIST consumes
          consumesname CDATA #REQUIRED
          eventtype    CDATA #REQUIRED >
```

consumesname

The **consumesname** attribute identifies the name associated with the consumes statement in idl.

eventtype

The **eventtype** attribute identifies the repository id of the event that the component expects to consume.

6.4.5.10 The containermanagedpersistence Element

Child element of **segment**.

A **containermanagedpersistence** element specifies attributes required by the container to manage the componentÕs persistent state using a PSS.

¥ **storagehome** indicates the type of abstract storage home.

- **pssimplementation** identifies a particular PSS implementation to be used, if not specified, then the default PSS is used, as determined by the container implementation.

Note Ð Issue 5499

- ~~**catalog** specifies the catalog type.~~
- **accessmode** specifies the access mode--read only or read-write.
- **psstransactionpolicy** specifies whether transactions are to be used or not and, if so, the isolation level.
- **params** is used to specify vendor specific parameters.

Note Ð Issue 5499

```
<!ELEMENT containermanagedpersistence
        ( storagehome
        , pssimplementation?
        ~~, catalog?~~
        , accessmode
        , psstransactionpolicy
        , params?
        ) >
```

6.4.5.11 The corbacomponent Element

The root element of this CORBA Component descriptor. See Section 6.4.5.1, ÒThe corbacomponent Root Element,Ó on page 6-21.

6.4.5.12 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 Ò.Ó. For example, Ò<corbaversion>3.0</corbaversion>Ó.

```
<!ELEMENT corbaversion (#PCDATA) >
```

6.4.5.13 The description Element

See Section 6.3.2.8, ÒThe description Element,Ó on page 6-7.

6.4.5.14 The emits Element

Child element of **ports**.

An **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. The **eventpolicy** allows the transaction policy of the event port to be specified.

```
<!ELEMENT emits
                              ( eventpolicy
                              , extension* ) >
<!ATTLIST emits
                              emitsname CDATA #REQUIRED
                              eventtype CDATA #REQUIRED >
```

The **emitsname** attribute identifies the name associated with the emits statement in idl.

The **eventtype** attribute identifies the repository id of the emitted events.

6.4.5.15 The entity Element

Child element of **componentkind**.

The **entity** component kind is described in Section 4.2.1, ÒComponent Containers,Ó on page 4-5.

```
<!ELEMENT entity
                              ( servant ) >
```

6.4.5.16 The eventpolicy Element

Note Ð Issue 5494

Child element of ~~**corbacomponent**~~, **consumes**, **emits**, **publishes**.

Event policies define the quality of service associated with the event ports of the component. The possible values are defined in Section 4.2.8, ÒEvents,Ó on page 4-12.

```
<!ELEMENT eventpolicy EMPTY>
<!ATTLIST eventpolicy
        policy ( normal | default | transaction ) #IMPLIED>
```

6.4.5.17 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.

6.4.5.18 The extension Element

Child element of **corbacomponent, componentfeatures, homefeatures**.

See Section 6.3.2.11, ÒThe extension Element,Ó on page 6-8.

6.4.5.19 The fileinarchive Element

See Section 6.3.2.12, ÒThe fileinarchive Element,Ó on page 6-9.

6.4.5.20 The homefeatures Element

Child element of **corbacomponent**.

The **homefeatures** element is used to describe a component home with respect to the homes that it inherits from and the **operationpolicies** of its interface.

```
<!ELEMENT homefeatures
          ( inheritshome?
           , operationpolicies?
           , extension* ) >
<!ATTLIST homefeatures
           name  CDATA #REQUIRED
          repid CDATA #REQUIRED >
```

The **name** attribute is the non-qualified name of the home.

The **repid** attribute is the fully qualified repository id of the home. **repid** is also used to refer to this component from elsewhere in the descriptor, for example from the **inheritshome** element.

6.4.5.21 The homeproperties Element

The **homeproperties** element specifies a default home property file. The format of the property file is described in Section 6.8, ÒProperty File Descriptor,Ó on page 6-62.

```
<!ELEMENT homeproperties
          ( fileinarchive
           ) >
```

6.4.5.22 *The homerepid Element*

Child element of **corbacomponent**.

homerepid identifies the repository id of the home of the component described by this descriptor. The home repository id also serves to point to the primary **homefeatures** element for the home within the descriptor, so as to distinguish it from inherited homes.

```
<!ELEMENT homerepid EMPTY >
<!ATTLIST homerepid
                                repid CDATA #IMPLIED >
```

repid is the fully qualified repository id of the component.

6.4.5.23 *The inheritscomponent Element*

Child element of **componentfeatures**.

The **inheritscomponent** element specifies an inherited component.

```
<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
   repid CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited component. It also serves to refer to the **componentfeatures** element of the inherited component elsewhere in the descriptor.

6.4.5.24 *The inheritshome Element*

Child element of **homefeatures**.

The **inheritshome** element specifies an inherited home.

```
<!ELEMENT inheritshome EMPTY>
<!ATTLIST inheritshome
   repid CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited home. It also serves to refer to the **homefeatures** element of the inherited home elsewhere in the descriptor.

6.4.5.25 *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 CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited interface. It is used to refer to the interface element of the inherited interface elsewhere in the descriptor.

6.4.5.26 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.

6.4.5.27 The interface Element

Child element of **corbacomponent**.

Specifies an interface that the component, either directly or through inheritance, provides, uses, or supports. The **operationpolicies** child element specifies default transaction policies and required security rights for uses of the interface.

```
<!ELEMENT interface
          ( inheritsinterface*
          , operationpolicies? ) >
<!ATTLIST interface
          name  CDATA #REQUIRED
          repid CDATA #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.

6.4.5.28 The interop Element

Child element of **corbacomponent**.

The **interop** element is used to specify whether this component interoperates with another component type by acting as a view for that type or having a view of that type.

```
<!ELEMENT interop EMPTY>
<!ATTLIST interop
        type CDATA #REQUIRED
        direction ( hasview | isview ) #REQUIRED
        descriptor CDATA  #REQUIRED >
```

The **type** attribute is the other component type (e.g., ÒEJB 1.1Ó).

The **direction** attribute says whether the CORBA component is a view for the other component type or the other way around.

The **descriptor** attribute references the descriptor file of the foreign component within the component archive.

6.4.5.29 The link Element

See Section 6.3.2.18, ÒThe link Element,Ó on page 6-11.

6.4.5.30 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.

6.4.5.31 The operation Element

Child element of **operationpolicies**.

The **operation** element is used to specify transaction and required security rights for a particular operation (or group of operations if name=Ó*Ó).

```
<!ELEMENT operation
        ( transaction?
         , requiredrights? ) >
<!ATTLIST operation
            name CDATA #REQUIRED >
```

The **name** attribute specifies the name of the operation. If the name is specified as Ò*****Ó, then the policies specified by this element apply to all operations in the particular scope in which the **operationpolicies** parent element is defined.

6.4.5.32 The operationpolicies Element

Child element of **componentfeatures, homefeatures, interface, provides,** and **supportsinterface**.

The **operationpolicies** element is used to specify a set of operation policies. It consists of a list of operation child elements that each may specify security or transaction policies of an operation or set of operations.

The scope of the **operationpolicies** element depends upon where it is specified. As a child of **componentfeatures** it specifies the policies for the component operations, such as the operations effecting facets, receptacles, and event ports. When used as a child of **homefeatures** it specifies the policies of the home interface operations. As a child of **interface** it specifies the operation policies for all uses of the particular interface. Operation policies set in a **supportsinterface** or **provides** element specify operation policies for a particular use of an interface. Note that operation policies set in **supportsinterface** or **provides** element supersede policies set in an **interface** element.

```
<!ELEMENT operationpolicies
          ( operation+ ) >
```

6.4.5.33 The param Element

Child element of **params**.

The **param** element is used to specify a name-value pair.

```
<!ELEMENT param EMPTY >
<!ATTLIST param
          name CDATA #REQUIRED
          value CDATA #REQUIRED >
```

The **name** attribute specifies the name.

The **value** attribute specifies the value.

6.4.5.34 The params Element

Child element of **containermanagedpersistence**.

The **params** element is used to specify a set of one or more name-value pairs.

```
<!ELEMENT params (param+) >
```

6.4.5.35 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 Ð Not all combinations of POA policies are valid. A good component packaging tool will not permit the user to specify invalid POA policy combinations. If however, an invalid combination of policies is used to configure the empty container, the container/POA should throw an exception.

6.4.5.36 *The ports Element*

Child element of **componentfeatures**.

The **ports** element describes what interfaces a component provides and uses, and what events it emits, publishes, and consumes. Any number of uses, provides, emits, publishes, and consumes elements can be specified in any order.

```
<!ELEMENT ports
        ( uses
        | provides
        | emits
        | publishes
        | consumes
        )* >
```

6.4.5.37 *The process Element*

Child element of **componentkind**.

The **process** component kind is described in Section 4.1.4, ÒComponent Categories,Ó on page 4-5.

```
<!ELEMENT process
         ( servant ) >
```

6.4.5.38 *The provides Element*

Child element of **ports**.

The **provides** element specifies an interface that is provided by the component.

The optional **operationpolicies** child element allows transaction policies and required rights to be specified for the provided interface. The policies specified here override any policies specified in the **interface** element, as identified by the **repid**.

```
<!ELEMENT provides
            ( operationpolicies?
            , extension* ) >
<!ATTLIST provides
           providesname CDATA #REQUIRED
           repid        CDATA #REQUIRED
           facettag     CDATA #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.

The **facettag** is the tag for the facet. This attribute is used in combination with the **segmentmember** element, defined in Section 6.4.5.48, ÒThe segmentmember Element,Ó on page 6-37, to associate a facet with a segment.

6.4.5.39 The pssimplementation Element

Child element of **containermanagedpersistence**.

The **pssimplementation** element identifies a particular vendorÕs PSS implementation.

```
<!ELEMENT pssimplementation EMPTY>
<!ATTLIST pssimplementation
           id CDATA #REQUIRED >
```

The **id** attribute identifies the particular PSS implementation.

6.4.5.40 psstransaction Element

Child element of **containermanagedpersistence**.

The **psstransaction** element is used to specify the PSS transactional policies associated with the entity or process component.

```
<!ELEMENT psstransaction (psstransactionisolationlevel?) >
<!ATTLIST psstransaction
           policy (TRANSACTIONAL|NON_TRANSACTIONAL) #REQUIRED >
```

6.4.5.41 psstransactionisolationlevel Element

Child element of **psstransaction.**

The **psstransactionisolationlevel** element is used to specify the transaction isolation level when persistent store access is transactional.

```
<!ELEMENT psstransactionisolationlevel EMPTY>
<!ATTLIST psstransactionisolationlevel
          level (  READ_UNCOMMITTED|READ_COMMITTED|
                   REPEATABLE_READ|SERIALIZABLE) #REQUIRED >
```

The **level** attribute identifies one of four isolation levels.

6.4.5.42 *The publishes Element*

Child element of **ports**.

A publishes port specifies an event that the component publishes. At deployment or creation time, the component will be connected to a channel by which it can be connected to consuming components. The **eventpolicy** allows the transaction policy of the event port to be specified.

```
<!ELEMENT publishes
                              ( eventpolicy
                              , extension* ) >
<!ATTLIST publishes
                              publishesname CDATA #REQUIRED
                              eventtype CDATA #REQUIRED >
```

The **publishesname** attribute identifies the name associated with the emits statement in idl.

The **event_type** attribute identifies the repository id of the published events.

6.4.5.43 *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
      | link
      ) >
<!ATTLIST repository
          type CDATA #IMPLIED >
```

The **type** attribute specifies the type of repository. Currently, the only predefined value for **type** is Ò**CORBA Interface Repository**.Ó

6.4.5.44 *The requiredrights Element*

Child element of **operation** and **security**.

The **requiredrights** element specifies a list of required rights. When used as a child of **operation**, the rights specified must belong to a rights family specified in the **security** element. When used as a child of **security** the list of rights specify the available rights in the rights family.

```
<!ELEMENT requiredrights
        ( right* ) >
```

6.4.5.45 The right Element

Child element of **requiredrights**.

The **right** element specifies a particular required right. The right must be a member of the rights family specified by the security element.

```
<!ELEMENT right
   ( description? ) >
<!ATTLIST right
   name CDATA #REQUIRED >
```

The **name** attribute is the name of the required right.

6.4.5.46 The security Element

Child element of **corbacomponent**.

The **security** element is an optional child element of **corbacomponent**; it is required whenever rights are assigned to component operations within the descriptor. It specifies the rights family assumed when defining component operation rights and the rights combinator required for interpretation of multiple required rights. The optional **requiredrights** element may be used to document the rights available in the rights family.

```
<!ELEMENT security
        ( requiredrights? ) >
<!ATTLIST security
        rightsfamily CDATA #REQUIRED
        rightscombinator (secallrights | secanyrights) #REQUIRED >
```

The **rightsfamily** attribute defines the rights family (for example, the ÒCORBAÓ rights family). The **rightscombinator** enumeration attribute defines the possible rights combinator describing the interpretation of multiple rights as defined by the CORBA Security Service.

6.4.5.47 The segment Element

Child element of **corbacomponent**.

The **segment** element describes a component segment. It consists of a list of one or more **segmentmember** child elements, indicating the facets that the segment supports, and a **containermanagedpersistence** element indicating that the persistent state of the segment is managed by the container. If the **containermanagedpersistence** element is not present, then the persistent state, if any, is managed by the component. Note that the **containermanagedpersistence** element is only employed for **entity** and **process** components.

```
<!ELEMENT segment
            ( segmentmember+
            , containermanagedpersistence?
            , extension*
            ) >
<!ATTLIST segment
            name      CDATA #REQUIRED
            segmenttag CDATA #REQUIRED >
```

name is the name of the segment.

segmenttag is the segmentÕs tag.

6.4.5.48 The segmentmember Element

Child element of **segment**.

The **segmentmember** element specifies a facet that is a member of a segment.

```
<!ELEMENT segmentmember EMPTY>
<!ATTLIST segmentmember
        facettag CDATA #REQUIRED >
```

The **facettag** attribute indicates the member facetÕs tag. It corresponds to a provided interface with the same facet tag elsewhere in the descriptor. (See the **provides** tag element in Section 6.4.5.38, ÒThe provides Element,Ó on page 6-33.)

6.4.5.49 The servant Element

Child element of **entity, process, session**.

Servant lifetime policies control the lifetime of the servant that implements a componentÕs 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 (component|method|transaction|container) #REQUIRED >
```

The possible values are defined in Section 4.2.5, ÒServant Lifetime Management,Ó on page 4-8.

6.4.5.50 The service Element

Child element of **componentkind**.

Specifies that the component is of the **service** category. The service component kind is described in Section 4.2.13.1, ÒThe Service Component,Ó on page 4-16.

```
<!ELEMENT service EMPTY >
```

6.4.5.51 The session Element

Child element of **componentkind**.

Specifies that the component is of the **session** category. The **session** component category is described in Section 4.2.13.2, ÒThe Session Component,Ó on page 4-17.

```
<!ELEMENT session
        ( servant ) >
```

6.4.5.52 The storagehome Element

Child element of **segment**.

The **storagehome** element specifies an abstract storage home type.

```
<!ELEMENT storagehome EMPTY>
<!ATTLIST storagehome
        id CDATA #REQUIRED >
```

The **id** attribute specifies the repository id of the abstract storage home.

6.4.5.53 The simple-link-attributes Entity

See Section 6.3.2.28, ÒThe simple-link-attributes Entity,Ó on page 6-14.

6.4.5.54 The supportsinterface Element

Child element of **componentfeatures**.

The **supportsinterface** element identifies an interface that the component supports, as defined in IDL.

The optional **operationpolicies** child element allows transaction policies and required rights to be specified for the supported interface. The policies specified here override any policies specified in the **interface** element, as identified by the **repid**.

```
<!ELEMENT supportsinterface
```

```
        ( operationpolicies?
        , extension* ) >
<!ATTLIST supportsinterface
        repid CDATA #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.

6.4.5.55 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 **policy** to **serialize** means that the container will serialize calls to the container.

Setting **policy** to **multithread** means that multiple threads of control can be active in the component at one time.

6.4.5.56 The transaction Element

Child element of **corbacomponent**.

The **transaction** element controls the way transactions are managed by the container for this component. Seven possible values can be selected by the component developer to provide maximum flexibility.

```
<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
   use (self-managed|not-supported|required|supports|requires-new|man-
datory|never) #REQUIRED >
```

If the transaction **use** attribute is set to **self-managed**, then it is assumed that the component will manage transactions on its own. Other **use** values indicate that transactions are to be managed by the container; the meaning of these values are defined in the container chapter, Section 4.2.6, ÒTransactions,Ó on page 4-9.

6.4.5.57 The unclassified Element

Child element of **componentkind**.

The **unclassified** element identifies that the component is of the unclassified sort. See Section 4.2.1, ÒComponent Containers,Ó on page 4-5 for more information on the unclassified component category.

```
<!ELEMENT unclassified
```

```
            ( poapolicies ) >
```

6.4.5.58 The uses Element

Child element of **ports**.

The **uses** element specifies an interface that is used by the component, as specified in a component IDL *uses* declaration.

```
<!ELEMENT uses ( extension* ) >
<!ATTLIST uses
            usesname CDATA #REQUIRED
            repid    CDATA #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.

6.5 Component Assembly Packaging

A component package is the vehicle for deploying a single component implementation. 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 be 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 Ð 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.

6.6 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 Ò.aarÓ extension.

6.7 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 Ò.cadÓ extension (Component Assembly Descriptor).

The assembly descriptor describes a component assembly. It consists of elements describing the components used in the assembly, connection information, and partitioning information.

A component instantiation is always relative to a home. A deployed home is called a home Òplacement.Ó

Component instantiations are connected by their *provides* and *uses* interfaces, or by their *emits*, *publishes*, 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 or publishes 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 Òmeta-inf.Ó

6.7.1 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 Chapter 7.

```
<!DOCTYPE componentassembly SYSTEM "componentassembly.dtd">

<componentassembly id="ZZZ123">
    <description>Example assembly"</description>
    <componentfiles>
        <componentfile id="A">
            <fileinarchive name="ca.csd"/>
        </componentfile>
        <componentfile id="B">
            <fileinarchive name="cb.csd"/>
        </componentfile>
        <componentfile id="C">
            <fileinarchive name="cc.csd">
                <link href="ftp://www.xyz.com/car/cc.car"/>
            </fileinarchive>
        </componentfile>
```

```
    <componentfile id="D">
        <fileinarchive name="cd.csd"/>
    </componentfile>
    <componentfile id="E">
        <fileinarchive name="ce.csd"/>
    </componentfile>
    <componentfile id="F">
        <fileinarchive name="cf.csd"/>
    </componentfile>
</componentfiles>

<partitioning>

    <homeplacement id="AaHome">
        <componentfileref idref="A"/>
        <componentinstantiation id="Aa"/>
    </homeplacement>

    <processcollocation cardinality="*">
        <usagename>Example process collocation</usagename>
        <impltype language="C++" />  <!-- optional -->
        <homeplacement id="BbHome">
            <componentfileref idref="B"/>
            <componentinstantiation id="Bb"/>
        </homeplacement>
        <homeplacement id="CcHome">
            <componentfileref idref="C"/>
            <componentinstantiation id="Cc"/>
        </homeplacement>
    </processcollocation>

    <hostcollocation cardinality="1">
        <usagename>Example host collocation</usagename>
        <processcollocation cardinality="*">
            <homeplacement id="DdHome">
                <componentfileref idref="D"/>
                <componentinstantiation id="Dd"/>
            </homeplacement>
            <homeplacement id="EdHome">
                <componentfileref idref="E"/>
                <componentinstantiation id="Ee"/>
            </homeplacement>
        </processcollocation>
        <homeplacement id="FfHome">
            <componentfileref idref="F"/>
            <componentinstantiation id="Ff"/>
        </homeplacement>
    </hostcollocation>

    <homeplacement id="AaaHome">
        <usagename>Example home for A components</usagename>
```

```
        <componentfileref idref="A"/>
        <componentimplref idref="an A impl"/>
        <homeproperties>
            <fileinarchive name="AHomeProperties.cpf"/>
        </homeproperties>
        <componentproperties>
            <fileinarchive name="defaultAProperties.cpf"/>
        </componentproperties>
        <registerwithhomefinder name="AaHome"/>

        <componentinstantiation id="Aaa">
            <usagename>Example component instantiation </usagename>
            <componentproperties>
                <fileinarchive name="AaaProperties.cpf"/>
            </componentproperties>
            <registercomponent>
                <registerwithnaming name="sink"/>
```

Note Ð Issue 5092

```
                <registerwithtrader>
                  <traderexport>
                    <traderservicetypename>
                        aTraderServiceTypeName
                    </traderservicetypename>
                    <traderproperties>
                        <traderproperty>
                            <traderpropertyname>ppm</traderpropertyname>
                            <traderpropertyvalue>10</traderpropertyvalue>
                        </traderproperty>
                        <traderproperty>
                            <traderpropertyname>weight</traderpropertyname>
                            <traderpropertyvalue>333</traderpropertyvalue>
                        </traderproperty>
                    </traderproperties>
                  </traderexport>
                </registerwithtrader>
            </registercomponent>
        </componentinstantiation>
    </homeplacement>

</partitioning>

<connections>
    <connectinterface>
        <usesport>
            <usesidentifier>abc</usesidentifier>
            <componentinstantiationref idref="Aa"/>
        </usesport>
        <providesport>
            <providesidentifier>abc</providesidentifier>
```

```
            <componentinstantiationref idref="Bb"/>
        </providesport>
    </connectinterface>
    <connectevent>
        <consumesport>
            <consumesidentifier>pqr</consumesidentifier>
            <componentinstantiationref idref="Aaa"/>
        </consumesport>
        <emitsport>
            <emitsidentifier>mno</emitsidentifier>
            <componentinstantiationref idref="Ee"/>
        </emitsport>
    </connectevent>
</connections>

</componentassembly>
```

6.7.2 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 Section 7.4, Òcomponentassembly.dtd,Ó on page 7-13.

6.7.2.1 The componentassembly Root Element

The **componentassembly** element is the root element of the component assembly descriptor. The **description** element is text describing the assembly. The **componentfiles** element lists the component files that are used in the assembly, the **partitioning** element describes how homes and components are to be deployed. The **connections** element describes how deployed components and homes are to be connected**.** The **extension** element can be used to add proprietary or experimental elements to the component assembly document.

```
<!ELEMENT componentassembly
        ( description?
        , componentfiles
        , partitioning
        , connections?
        , extension*
        ) >
<!ATTLIST componentassembly
        id ID #REQUIRED
        derivedfrom CDATA #IMPLIED >
```

The **id** attribute is a DCE UUID that uniquely identifies the assembly.

The **derivedfrom** attribute is used to point to an assembly from which this assembly was derived. The **derivedfrom** attribute contains the id of the source assembly.

Note Ð The **derivedfrom** attribute is for a deployment tool that wants to create a copy of an assembly descriptor and archive to describe an actual deployment; it maintains the relationship between the ÒcloneÓ and the original. The new assembly descriptor would have the destination addresses for each placement and collocation defined; and collocations with non-ordinal cardinality in the original assembly would be copied to one or more collocations, with singular cardinality, in the derived assembly. The new archive file might prune constituent component archive files to contain single implementations to facilitate copying component implementations to target deployment hosts.

6.7.2.2 *The codebase Element*

See Section 6.3.2.4, ÒThe codebase Element,Ó on page 6-6.

6.7.2.3 *The componentfile Element*

The **componentfile** element refers to a component software descriptor containing information regarding a component and home implementation. **componentfile** elements are referenced by **homeplacement** elements.

componentfile contains either a **fileinarchive, link**, or **codebase** element.

```
<!ELEMENT componentfile
            ( fileinarchive
            | codebase
            | link
            ) >
<!ATTLIST componentfile
            id   ID    #REQUIRED
            type CDATA #IMPLIED >
```

The **id** attribute must uniquely identify the **componentfile** element within the descriptor.

The optional **type** attribute specifies the type of component file. If unspecified, then the file is assumed to be CORBA component. An example use of the type attribute would be to specify an EJB component file, where **type**=ÓEJB 1.1Ó.

6.7.2.4 *The componentfileref Element*

The **componentfileref** element refers to a particular **componentfile** element in the **componentfiles** block.

```
<!ELEMENT componentfileref EMPTY >
<!ATTLIST componentfileref
           idref CDATA #REQUIRED >
```

The **idref** attribute corresponds to a unique **componentfile id** attribute.

6.7.2.5 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 specified.

Each component file is uniquely identified for reference elsewhere in the descriptor. Multiple component instances may refer to a single component file.

```
<!ELEMENT componentfiles
        ( componentfile+
        ) >
```

6.7.2.6 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** element **id** in the component descriptor. The **componentimplref** is optional if there is only one implementation in the component file. Or it may be set at deployment time depending on the type of platform that the component is deployed to.

6.7.2.7 The componentinstantiation Element

The **componentinstantiation** element describes a particular instantiation of a component relative to a home placement. The **componentinstantiation** element is a direct child of the **homeplacement** element.

The **usagename** child element is used to specify a name for the placement, possibly for display in a tool. The **componentproperties** element refers to a property file associated with this instantiation. It is used to configure the component once it is created and after the home sets initial property values (as specified in the **homeplacement componentproperties** element). The **registercomponent** element instructs the installation process to register the component or its provided interfaces with a naming service or trader.

```
<!ELEMENT componentinstantiation
        ( usagename?
        , componentproperties?
        , registercomponent*
        , extension*
        ) >
<!ATTLIST componentinstantiation
        id      ID   #REQUIRED >
```

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.

6.7.2.8 The componentinstantiationref Element

The **componentinstantiationref** element refers to a particular **componentinstantiation** element in the assembly descriptor.

```
<!ELEMENT componentinstantiationref EMPTY >
<!ATTLIST componentinstantiationref
    idref CDATA #REQUIRED >
```

The **idref** attribute corresponds to a unique **componentinstantiation** id attribute.

6.7.2.9 The componentproperties Element

The **componentproperties** element specifies a property file for a home. 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 6.8, ÒProperty File Descriptor,Ó on page 6-62.

When the **componentproperties** element is specified as part of a **homeplacement** element, then the properties are used to configure each component created through that home. When **componentproperties** is specified as part of a **componentinstantiation** element, the properties are used to configure that single instantiation. If component properties are set on both a **homeplacement** and an associated **componentinstantiation**, then the component will be configured first by the **homeplacement** component properties and then by the **componentinstantiation** component properties.

```
<!ELEMENT componentproperties
          ( fileinarchive
          | codebase
           ) >
```

6.7.2.10 The componentsupportedinterface Element

Specifies a component with a *supports* interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element. The component is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT componentsupportedinterface
          ( componentinstantiationref
          | findby
          )>
```

6.7.2.11 The connectevent Element

Note Ð Issue 5093

The **connectevent** element is used in the **connections** element to specify a connection from a *consumes* port, of one component, or an existing interface to an *emits* or *publishes* port of another component.

The **consumesport** element identifies a component and associated consumes port. The **emitsport** element identifies a component associated emits port. The **publishesport** element identifies a component and associated **publishesport**.

```
<!ELEMENT connectevent
          (    ( consumesport
               | existinginterface
               )
          ,    ( emitsport
               | publishesport
               )
          ) >
<!ATTLIST connectevent
    id ID #IMPLIED >
```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor; however, someone (or a tool) might want to use it to refer to a particular **connectevent** element.

6.7.2.12 The connecthomes Element

The **connecthomes** element is used to specify a connection between a **proxyhome** and another home.

The **proxyhome** element refers to the proxy home. The **destinationhome** element refers to the home to which the proxy home will be connected. The destination home can be either another proxy home or an actual home.

```
<!ELEMENT connecthomes
          ( proxyhome
          , destinationhome
          ) >
<!ATTLIST connecthomes
          id ID #IMPLIED >
```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor; however, someone (or a tool) might want to use it to refer to a particular **connecthomes** element.

6.7.2.13 The connectinterface Element

The **connectinterface** element is used to connect a componentÕs *uses* port to an interface. The interface may be a provided or supported interface of another component, it may be an existing interface (other than those provided by components in the assembly), or it may be a home interface.

The **usesport** element identifies the component and port where the connection is to be made. The **providesport** element identifies a component and provides port. The **componentsupportedinterface** element identifies a component that has a supported interface that will satisfy the uses port. The **existinginterface** element identifies a way to find an existing interface that will satisfy the *uses*. The **homeinterface** element identifies a **homeinterface** that the uses port requires.

```
<!ELEMENT connectinterface
       ( usesport
     ,     ( providesport
                 | componentsupportedinterface
                 | existinginterface
                 | homeinterface
                 )
     ) >
<!ATTLIST connectinterface
    id ID #IMPLIED >
```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor; however, someone (or a tool) might want to use it to refer to a particular **connectinterface** element.

6.7.2.14 The connections Element

The connections element is used to satisfy component uses and consumes dependencies and to connect homes. The **connectinterface** element is used to connect component *uses* ports to interfaces. The **connectevent** element is used to connect a components *consumes* port to event producers. The **connecthomes** element is used to connect a proxy home to another home.

```
<!ELEMENT connections
    ( connectinterface
    | connectevent
    | connecthomes
    | extension
    )* >
```

Note Ð If a **componentinstantiation** involved in 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. That is, the connection will be made for each instantiation of the component.

6.7.2.15 The consumesidentifier Element

A child element of **consumingcomponent**, **consumesidentifier** identifies which *consumes* ÒportÓ on the component is to participate in the relationship. The type of the consumes event must match the type of the connected emits or publishes event.

```
<!ELEMENT consumesidentifier ( #PCDATA ) >
```

6.7.2.16 The consumesport Element

Specifies the event-consuming side of an event connection relationship. The **consumesidentifier** child element identifies the particular *consumes* port. The component with this consumes port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT consumesport
        ( consumesidentifier
        ,    ( componentinstantiationref
             | findby
             )
        )>
```

6.7.2.17 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 ) >
```

6.7.2.18 The destination Element

The **destination** element is used to record where a **homeplacement**, **executableplacement**, **hostcollocation**, or **processcollocation** is to be (or has been) deployed. The format of the destination string is determined by a particular deployment tool.

```
<!ELEMENT destination ( #PCDATA ) >
```

6.7.2.19 The destinationhome Element

Identifies a home to be connected to by a proxy home. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, naming service, or trader, or using a stringified object reference.

```
<!ELEMENT destinationhome
```

```
        ( homeplacementref
        | findby
        ) >
```

6.7.2.20 The emitsidentifier Element

The **emitsidentifier** identifies an emits ÒportÓ on a component. The identifier corresponds to an emits identifier specified in IDL.

```
<!ELEMENT emitsidentifier ( #PCDATA ) >
```

6.7.2.21 The emitsport Element

Specifies the event-emiting side of an event connection relationship. The **emitsidentifier** child element identifies the particular *emits* port. The component with this emits port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT emitsport
                            ( emitsidentifier
                            ,    ( componentinstantiationref
                                 | findby
                                 )
                            )>
```

6.7.2.22 The executableplacement Element

This **executableplacement** element describes a deployment of an executable. The **executableplacement** element may be a direct child of the **partitioning** element, which states that it has no collocation constraints; or it may be a child element of the **hostcollocation** element.

The **usagename** child element is used to specify a name for the placement, possibly for use in a tool. The **componentfileref** element specifies the component file. The **componentimplref** element refers to a specific implementation in the **softpkg** descriptor. Note that the implementation referred to by **componentimplref** must have a code type of Ò**Executable**.Ó The **invocation** element specifies any arguments with which the executable should be invoked. The **destination** element is used to record where the **executableplacement** is to be deployed.

```
<!ELEMENT executableplacement
        ( usagename?
        , componentfileref
        , componentimplref?
        , invocation?
        , destination?
        , extension*
```

```
            ) >
<!ATTLIST executableplacement
            id          ID    #REQUIRED
            cardinality CDATA "1" >
```

The **id** attribute is a unique identifier within the assembly descriptor for the **executableplacement**.

The **cardinality** attribute specifies how many instantiations of this executable may be deployed. Possible values for cardinality are a specific number, a Ò+Ó to specify 1 or more, or a Ò*Ó to specify 0 or more. The default cardinality is Ò1.Ó

6.7.2.23 *The existinginterface Element*

Note Ð Issue 5093

Specifies an interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element, or to an emits or publishes port within a **connectevent** element. The **findby** element points to an existing interface that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT existinginterface
            ( findby ) >
```

6.7.2.24 *The extension Element*

See Section 6.3.2.11, ÒThe extension Element,Ó on page 6-8.

6.7.2.25 *The fileinarchive Element*

Note Ð Issue 5495

~~See Section 6.3.2.11, ÒThe extension Element,Ó on page 6-8.~~

See Section 6.3.2.12, ÒThe fileinarchive Element,Ó on page 6-9.

6.7.2.26 *The findby Element*

The **findby** element is used to resolve a connection between two components. It tells the installation agent how to locate a party, usually a component, interface, or home involved in the relationship. In the simplest case, the installer will know where the item is because it was the one responsible for installing it. But if the item to be located already exists in the installation environment, the installer must know how to find it. It could locate a component in a naming service, in a trader, a home finder, or by a stringified object reference. The purpose of the findby element is to provide such information.

The **namingservice** element specifies a naming service name. The **stringifiedobjectref** element is a stringified IOR for the item. The **traderquery** is a query for locating the item in a trader. The **homefinder** is a name to look up a home in a home finder.

```
<!ELEMENT findby
        ( namingservice
        | stringifiedobjectref
        | traderquery
        | homefinder
        | extension
        ) >
```

6.7.2.27 The homefinder Element

The **homefinder** element is used to indicate a home finder name for a home.

```
<!ELEMENT homefinder EMPTY >
<!ATTLIST homefinder
        name CDATA #REQUIRED >
```

The **name** attribute specifies the name of the home as registered with the home finder. Home finders are defined in Section 1.8, ÒHome Finders,Ó on page 1-42.

6.7.2.28 The homeinterface Element

Specifies a home with an interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, a naming service or trader, or using a stringified object reference.

```
<!ELEMENT homeinterface
        ( homeplacementref
        | findby
        ) >
```

6.7.2.29 The homeplacement Element

This **homeplacement** element describes a particular deployment of a component home. The **homeplacement** element may be a direct child of the **partitioning** element which states that it has no collocation constraints; or it may be a child element of the **hostcollocation** or **processcollocation** elements that states specific host or process collocation constraints.

The **usagename** child element is used to specify a name for the placement, possibly for use in a tool. The **componentfileref** element specifies the component file. The **componentimplref** element refers to a specific implementation in the component file.

The **homeproperties** element refers to a state file associated with the home placement; it is used to configure the home after it is created. The **componentproperties** element refers to a property file used to configure all components created through the home. The **registerwithhomefinder** element instructs the installation process to register the home with the home finder. The **registerwithnaming** element instructs the installation process to register the home with a naming service. The **registerwithtrader** element instructs the installation process to register the home with a trader service. The **componentinstantiation** element instructs the installation agent to create a component using this home. The **destination** element is used to record where the **homeplacement** is to be deployed, if designated.

```
<!ELEMENT homeplacement
        ( usagename?
        , componentfileref
        , componentimplref?
        , homeproperties?
        , componentproperties?
        , registerwithhomefinder*
        , registerwithnaming*
        , registerwithtrader*
        , componentinstantiation*
        , destination?
        , extension*
        ) >
<!ATTLIST homeplacement
        id          ID    #REQUIRED
        cardinality CDATA "1" >
```

The **id** attribute is a unique identifier within the assembly descriptor for the **homeplacement**. The **id** is used to refer to the home 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 Ò+Ó to specify 1 or more, or a Ò*Ó to specify 0 or more. The default cardinality is Ò1.Ó

Note that if the **cardinality** is greater than 1 and there are any connections to this **homeplacement**, then connections will be made to each instance of the deployed home.

6.7.2.30 The homeplacementref Element

The **homeplacementref** element refers to a particular **homeplacement** element in the assembly descriptor.

```
<!ELEMENT homeplacementref EMPTY >
<!ATTLIST homeplacementref
    idref CDATA #REQUIRED >
```

The **idref** attribute corresponds to a unique **homeplacement** id attribute.

6.7.2.31 The homeproperties Element

The **homeproperties** element specifies a property file for a home. The properties are used to configure the home when it is created. 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 6.8, ÒProperty File Descriptor,Ó on page 6-62.

```
<!ELEMENT homeproperties
          ( fileinarchive
          | codebase
          ) >
```

6.7.2.32 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**, **homeplacement**, and **executableplacement** 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?
         ,    ( homeplacement
              | executableplacement
              | processcollocation
              | extension
              )+
         , destination?
         ) >
<!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 Ò+Ó to specify 1 or more, or a Ò*Ó to specify 0 or more. The default cardinality is Ò1.Ó

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.

6.7.2.33 The impltype Element

Placeholder for future version.

6.7.2.34 The invocation Element

The **invocation** element is used to specify invocation arguments for an executable placement.

```
<!ELEMENT invocation EMPTY >
<!ATTLIST invocation
        args CDATA    #REQUIRED >
```

The **args** attribute is a string containing the arguments to be used in invoking the executable. Note that **args** is just the arguments to the executable, it does not include the executable name.

6.7.2.35 The link Element

See Section 6.3.2.18, ÒThe link Element,Ó on page 6-11.

6.7.2.36 The namingservice Element

The naming service element is used to indicate that a component or interface should be found using a naming service.

```
<!ELEMENT namingservice EMPTY >
<!ATTLIST namingservice
        name CDATA #REQUIRED >
```

The name attribute specifies the naming service name to look up.

6.7.2.37 The partitioning Element

Component partitioning specifies a deployment pattern of homes and components to generic processes and hosts. The pattern is expressed via collocation constraints.

A particular usage of a component is always relative to a component home. Uses of component homes are recognized in the assembly as home placements. A home placement, and component instantiations relative to that home, may be collocated with other home placements and component instantiations in a process. Processes and home placements may be collocated within a logical host. A home placement that is not part of a process or host collocation may be deployed without constraint.

An executable placement is the placement of a particular executable. It may be partitioned without constraint or as part of a host collocation.

Within a **partitioning** element, **homeplacement**, **executableplacement**, and collocation constraints are specified. The **homeplacement** child element specifies a freely deployable home. The **executableplacement** element specifies a freely deployable executable. The **processcollocation** and **hostcollocation** child elements are used to group **homeplacement** together into deployable units.

A **homeplacement** 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. Home placements, executable placements, process collocations, and host collocations all have an associated cardinality. The default cardinality is Ò1.Ó An ordinal cardinality of 1 or greater mandates that the deployable unit must be instantiated that many times, cardinality of Ò+Ó indicates 1 or more, and Ò*Ó indicates zero or more.

```
<!ELEMENT partitioning
        ( homeplacement
        | executableplacement
        | processcollocation
        | hostcollocation
        | extension
        )* >
```

6.7.2.38 The processcollocation Element

The **processcollocation** element specifies a group of home and associated component instantiations that are to be deployed together to a single process. The child elements are an optional **usagename**, an optional **impltype**, and a list of **homeplacement** 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.

Note Ð Issue 5496

```
<!ELEMENT processcollocation
        ( usagename?
        , impltype?
        ,    ( homeplacement
             | extension
             )+
        , destination?
        ) >
<!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 Ò+Ó to specify 1 or more, or a Ò*Ó to specify 0 or more. The default cardinality is Ò1.Ó

Note that if the **cardinality** is greater than 1, and there are connections to components and homes within the **processcollocation**, then connections will be made to corresponding components or component homes within each instance of the collocation.

6.7.2.39 The providesidentifier Element

The **providesidentifier** identifies a provides ÒportÓ on a component. The identifier corresponds to a provides identifier specified in component IDL.

<!ELEMENT providesidentifier (#PCDATA) >

6.7.2.40 The providesport Element

Specifies the interface providing side of an interface connection relationship. The **providesidentifier** child element identifies the particular *provides* port. The component with this provides port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT providesport
        ( providesidentifier
        ,    ( componentinstantiationref
             | findby
             )
         )>
```

Note Ð Issue 5497

6.7.2.41 The proxyhome Element

Identifies a proxy home that is to be connected to another home. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, naming service, or trader, or using a stringified object reference.

<!ELEMENT ~~remote~~proxyhome
(homeplacementref
| findby
) >

6.7.2.42 The publishesidentifier Element

The **publishesidentifier** identifies a publishes ÒportÓ on a component. The identifier corresponds to the identifier specified in IDL for the publishes port.

<!ELEMENT publishesidentifier (#PCDATA) >

6.7.2.43 The publishesport Element

Specifies the event-publishes side of an event connection relationship. The **publishesidentifier** child element identifies the particular *publishes* port. The component with this publishes port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT publishesport
                              ( publishesidentifier
                              ,    ( componentinstantiationref
                                   | findby
                                   )
                              )>
```

6.7.2.44 The registercomponent Element

Note Ð Issue 5092

The **registercomponent** element is used to specify that a component, a provided interface, or an ~~published~~ event consumer interface should be registered with a naming service or trader.

If a~~n emitsidentifier,~~ **providesidentifier**~~,~~ or **~~publishes~~consumesidentifier** is specified, then that element is registered. If none of the above are specified, then it is implied that the component itself is to be registered.

Registration may be through a naming service or trader. The **registerwithnaming** element specifies a naming service registration and **registerwithtrader** specifies a trader registration. The interface, event, or component registration may be registered with both a naming service and a trader, multiple times. At least one registration must take place.

```
<!ELEMENT registercomponent
          (    ( ~~emitsidentifier~~
               ~~|~~providesidentifier
               | ~~publishes~~consumeridentifier
               )?
          ,    ( registerwithnaming
               | registerwithtrader
               )+
          ) >
```

6.7.2.45 The registerwithhomefinder Element

The **registerwithhomefinder** element tells the installer to register a component home with the home finder.

```
<!ELEMENT registerwithhomefinder EMPTY >
<!ATTLIST registerwithhomefinder
    name CDATA #REQUIRED >
```

The **name** attribute is the name to register the home with in the home finder.

6.7.2.46 *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 #IMPLIED >
```

The **name** attribute is the naming service name. If the name is not specified, it will be determined at deployment time, possibly with interaction with the user.

6.7.2.47 *The registerwithtrader Element*

The **registerwithtrader** element tells the installer to register a component instance or home with a trader after it is created.

Note Ð Issue 5092

```
<!ELEMENT registerwithtrader
        ( trader~~properties~~export ) >
<!ATTLIST registerwithtrader
        tradername CDATA #IMPLIED >
```

Note Ð Issue 5497

~~6.7.2.48 *The proxyhome Element*~~

~~Identifies a proxy home that is to be connected to another home. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, naming service, or trader, or using a stringified object reference.~~

```
~~<!ELEMENT remotehome~~
~~——      ( homeplacementref~~
~~—       | findby~~
~~——      ) >~~
```

6.7.2.49 The stringifiedobjectref Element

The **stringifiedobjectref** element is used to locate a component by its object reference.

```
<!ELEMENT stringifiedobjectref ( #PCDATA ) >
```

6.7.2.50 Trader elements

The trader elements are used to register a home, component or interface with a trader and to find a home, component, or interface 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 ( #PCDATA) >

<!ELEMENT traderpreference ( #PCDATA ) >

<!ELEMENT traderproperties
    ( traderproperty+ ) >

<!ELEMENT traderproperty
    ( traderpropertyname
    , traderpropertyvalue
    ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ( #PCDATA ) >

<!ELEMENT traderquery
    ( traderservicetypename
    , traderconstraint
    , traderpreference?
    , traderpolicy*
    , traderspecifiedprop*
    ) >
```

```
<!ELEMENT traderservicetypename ( #PCDATA ) >
```

```
<!ELEMENT traderspecifiedprop ( #PCDATA ) >
```

Note Ð Refer to the OMG Trading Object Service specification (formal/00-06-27) for additional information.

6.7.2.51 The usagename Element

A user defined ÒfriendlyÓ name.

```
<!ELEMENT usagename ( #PCDATA ) >
```

6.7.2.52 The usesidentifier Element

A child element of **usesport**, **usesidentifier** identifies which uses ÒportÓ 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 ) >
```

6.7.2.53 The usesport Element

Specifies the interface using side of an interface connection relationship. The **usesidentifier** child element identifies the particular *uses* port. The component with this uses port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT usesport
        ( usesidentifier
        ,    ( componentinstantiationref
             | findby
             )
        )>
```

6.8 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 packaged 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 Ò.cpfÓ (Component Property File).

6.8.1 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 4 major elements: **simple**, **sequence**, **struct**, and **valuetype**. The **simple** element describes a single primitive idl type. The **sequence** element corresponds to an IDL sequence, the **struct** element corresponds to an IDL struct, and the **valuetype** element corresponds to an IDL valuetype.

Note Ð If the user of the property file does not have static information about the types specified in the property file, then it will likely need to construct the type into a **DynAny**.

6.8.2 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 Section 7.3, Òproperties.dtd,Ó on page 7-11.

6.8.2.1 The properties Root Element

The **properties** element is the root element of the properties document. It contains an optional description and any combination of **simple**, **sequence**, **struct**, and **valuetype** elements.

```
<!ELEMENT properties
        ( description?
        ,    ( simple
             | sequence
             | struct
             | valuetype
             )*
        ) >
```

6.8.2.2 The choice Element

```
<!ELEMENT choice ( #PCDATA ) >
```

The **choice** element is used to specify a valid simple property value.

6.8.2.3 The choices Element

```
<!ELEMENT choices ( choice | range )+ >
```

The **choices** element is a list of one or more choice or range elements.

6.8.2.4 The defaultvalue Element

```
<!ELEMENT defaultvalue ( #PCDATA ) >
```

The **defaultvalue** element is used to specify a default simple property value.

6.8.2.5 The description Element

```
<!ELEMENT description ( #PCDATA ) >
```

The **description** element is used to provide a description of its enclosing element.

6.8.2.6 *The properties Element*

The root element of the properties file. See Section 6.8.2.1, ÒThe properties Root Element,Ó on page 6-64.

6.8.2.7 *The range Element*

```
<!ELEMENT range (value, value) >
```

The **range** element is a set of two **value** elements that define a specific range of valid simple property values. The order of the range limits; that is, (min, max) or (max, min) is not implied.

6.8.2.8 *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?
        ) >
```

Note Ð Issue 5429

```
<!ATTLIST simple
        name CDATA #IMPLIED
        type (  boolean
                | char
                | double
                | float
                | short
                | long
                | objref
                | octet
                ~~| short~~
                | string
                | ulong
                | ushort
                | longlong
```

```
                | ulonglong
                | longdouble
                | wchar
                | wstring
                | fixed
        ) #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 Ð 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.

6.8.2.9 *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, a sequence of valuetypes, 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*
             | valuetype*
             )
        ) >


<!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.

6.8.2.10 The struct Element

The **struct** element corresponds to an IDL structure. It may be composed of simple properties, sequences, structs, or other valuetypes.

```
<!ELEMENT struct
        ( description?
        ,   ( simple
            | sequence
            | struct
            | valuetype
            )*
        ) >
<!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.

6.8.2.11 The value Element

The **value** element is used to specify a simple value.

```
<!ELEMENT value ( #PCDATA ) >
```

6.8.2.12 The valuetype Element

The **valuetype** element is used to specify an IDL valuetype. It may be composed of simple properties, sequences, structs, or other valuetypes.

```
<!ELEMENT valuetype
        ( description?
        ,   ( simple
             | sequence
            | struct
            | valuetype
            )*
```

```
        ) >
<!ATTLIST valuetype
        name CDATA #IMPLIED
        type CDATA #REQUIRED
        primarykey (true | false) ÒfalseÓ >
```

name

The **name** attribute specifies the name of the valuetype attribute as it appears in IDL. The **name** attribute is required, except when the valuetype property is used in a sequence.

type

The **type** attribute specifies the repository id of the corresponding IDL valuetype.

primarykey

The **primarykey** attribute indicates whether or not the valuetype property provides the state information for an entity component primary key with a repository id given by the **type** attribute.

6.9 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 aim of deployment is to install and Òhook-upÓ a logical component topology to a physical computing environment. The component topology is specified by an assembly package, or in the degenerate case, an individual component package.

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 component homes and components on particular hosts. The mapping for doing so was determined in step 1.
4. Connect components as specified in the assembly descriptor’s connect blocks.

A stand-alone component package may be deployed as well as assembly packages. 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.

6.9.1 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.

The following class diagram and scenario represents a deployment architecture.

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 69-542).

Note Ð ~~Of the interfaces described below, only **ComponentInstallation**, **AssemblyFactory**, and **Assembly** are required by this specification; the other interfaces are included for illustrative purposes and to support an end-to-end scenario.~~

6.9.1.1 Deployment Architecture

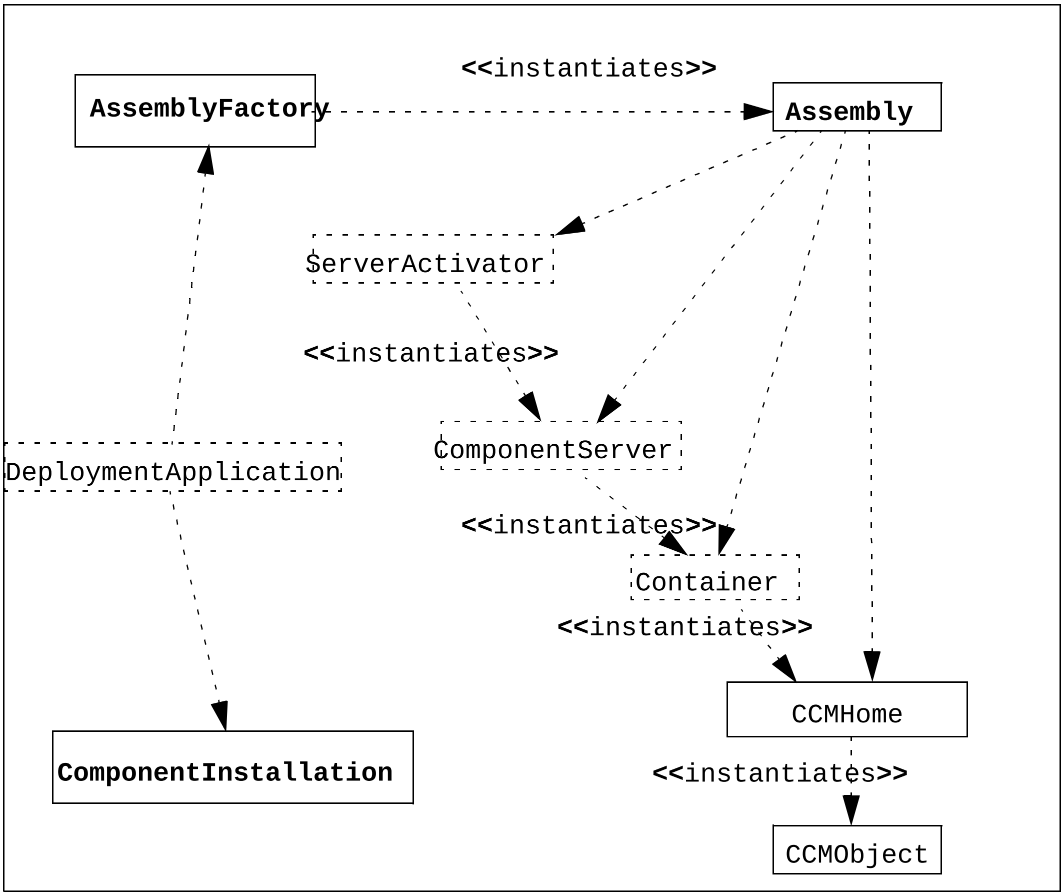


Figure 6-1 Deployment Architecture

6.9.1.2 Packages and Locations

References are made above and below to assembly and component *packages* and their *location* with some degree of intentional ambiguity to allow for future interpretations of a *package* and its *location*. At a minimum a *package* can be interpreted as a package descriptor or an archive that contains a package descriptor as described in Section 6.1, ÒIntroduction,Ó on page 6-1. A *location* can be interpreted as a URI or absolute path to a package or other file.

6.9.1.3 Deployment Scenario

The steps in deploying and activating a component assembly could unfold as follows.

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 and connectivity of the assembly.

2. Next the component implementations are installed on the platforms where they are expected to be used. The deployment application calls *install* on the **ComponentInstallation** object, passing the component implementation **id** and a string denoting the location of the component package. If the component has not already been installed on the target platform, then the **ComponentInstallation** object retrieves the component package and optionally makes it available in the local environment (component implementations and other needed files can be retrieved Òjust in timeÓ by a **ComponentInstallation** object if configured to do so).

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 **AssemblyFactory** is passed a string denoting the location of an assembly package. If necessary, the **AssemblyFactory** brings the assembly package into the local environment and makes its location known to the **Assembly** object.

4. The assembly object uses the assembly package as a recipe for creating the assembly. The assembly package 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 home and component 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 accomplished with the help of a set of objects on each host. These are **ServerActivator**, **ComponentServer**, **Container**, and the **CCMHome**.

6. The **Assembly** object first calls the **ServerActivator** on the target host to create a component server. There is one instance of the **ServerActivator** object on each host where components could be installed. The **Assembly** object creates the component server by calling the **create_component_server** operation on the **ServerActivator** object. This operation creates an empty server process and returns a reference to the **ComponentServer** object of the newly created process.

7. A **ComponentServer** object 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 list of container configuration values. 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 homes 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.

9. In order to create the home, the **Container** must load the DLL, shared object file, or .class file into the container process. To locate the component implementation, the container calls the **get_implementation** operation of the **ComponentInstallation** object. It passes in the **id** of the component implementation and is returned the location of the component implementation. The container then loads the implementation DLL, shared object file, or .class file that can be used to instantiate a home object. The home object reference is then returned to the **Assembly** object.

10. The **Assembly** object uses the component's home object to create a component instance. If the component category is *service*, *session*, or *process* an instance is created by calling **create_component** on the home reference. If the component category is *entity*, then a component instance is created by calling **create** on the home reference and passing in an instance of a value type derived from **Components::PrimaryKeyBase**. The primary key value type instance can be created using information supplied in a component property file. Both the **create** operation and the **create_component** return a **CCMObject** object reference.

11. If applicable, a configurator is applied to the component.

12. Once all of the components are installed, the **Assembly** object connects components in the assembly based on the information in the connect blocks of the assembly descriptor. It does so by calling the receptacle and event connection operations on the **CCMObject** references.

13. Following the successful consummation of each connection in the assembly, the **Assembly** object may call **configuration_complete** on each object in the assembly to signal that all of its configuration values have been set and all initial connections have been fixed.

Note Ð The operation configuration_complete should be called on components according to their interconnection dependencies. For instance, an active (containing a thread) component should not start before all the components connected to via its connection ports start, else the active component could use other components that have not yet started (i.e., on which configuration_complete has not yet been called). If circular dependencies exist between connected components then no assumption can be made as to the ÒbestÓ order that the configuration_complete operation should called on dependent components.

6.9.1.4 Configuration

Several operations defined below include argument parameters of type **ConfigValues** (described in Section 1.10.1.2, ÒThe StandardConfigurator interface,Ó on page 1-48). These configuration parameters are name value pairs that are used to pass configuration information in a standard way. Default configuration value names, if specified, are provided below with the corresponding operation that uses them. Operations that accept a **ConfigValues** parameter may raise an InvalidConfiguration exception if a configuration value passed by a calling procedure is invalid or unexpected. The relevant types and default list of reasons for invalid configuration name value pairs follow:

```
typedef FailureReason InvalidConfigurationReason;

const InvalidConfigurationReason UnknownConfigValueName = 0;
const InvalidConfigurationReason InvalidConfigValueType = 1;
const InvalidConfigurationReason ConfigValueRequired = 2;
const InvalidConfigurationReason ConfigValueNotExpected = 3;

exception InvalidConfiguration
{
    InvalidConfigurationReason reason;
    FeatureName name;
};
```

The **reason** field of the InvalidConfiguration exception could take the following **InvalidConfigurationReason** values:

- **UnknownConfigValueName** if the name of a configuration value is unknown by the called operation.
- **InvalidConfigValueType** if the configuration value is known but the type of the value is incorrect.
- **ConfigValueRequired** if the caller did not pass in a required configuration value.
- **ConfigValueNotExpected** if the configuration value is correct but not expected in the context of the called operation.

Note Ð This specification does not exhaustively define what the configuration values are (i.e., some of these may be specific to an ORB vendor). To provide deployment interoperability a future RFP should be submitted to standardize the configuration values once consensus among vendors will be established.

6.9.2 ComponentInstallation Interface

The **ComponentInstallation** object is used to install, query, and remove component implementations on a single platform. There is at most one **ComponentInstallation** object per host.

It is intended that this interface be general enough to encompass a wide range of underlying implementations, as the **ComponentInstallation** interface may be implemented on top of a vendor specific implementation repository.

Note that it is entirely possible for **ComponentInstallation** objects to forego physical installation of component packages and to delay actual location of and use of component implementations until needed by a container; that is, when a container calls the **get_implementation** operation of **ComponentInstallation**). This delayed location and use of implementations can reduce allocation of ÒlocalÓ physical storage and enable the use of distributed or remote implementation repositories or acquisition of implementations from services such as the Naming Service and Trader Service.

```
typedef string UUID;
typedef string Location;

exception UnknownImplId { };
exception InvalidLocation { };
exception InstallationFailure { FailureReason reason; };

interface ComponentInstallation {
      void install(in UUID implUUID,
             in Location component_loc)
      raises (InvalidLocation, InstallationFailure);
      void replace(in UUID implUUID,
            in Location component_loc)
      raises (InvalidLocation, InstallationFailure);
      void remove(in UUID implUUID)
      raises (UnknownImplId, RemoveFailure);
      Location get_implementation(in UUID implUUID)
      raises(UnknownImplId, InstallationFailure);
};
```

install

The **install** operation installs a component on the particular host on which the **ComponentInstallation** object resides. The **component_loc** parameter points to the location of the component package. The **implUUID** refers to a particular implementation within the component package. The operation raises an InvalidLocation exception if the **component_loc** parameter does not denote a valid package location. The operation raises an InstallationFailure exception if the package denoted by the **component_loc** parameter could not be properly installed on the host on which the **ComponentInstallation** object resides.

replace

The **replace** operation replaces a component package previously installed. The **component_loc** parameter points to the location of a component package. The **implUUID** refers to a particular implementation within the component package. The operation raises an **InvalidLocation** exception if the **component_loc** parameter does not denote a valid package location. The operation raises an **InstallationFailure** exception if the package denoted by the **component_loc** parameter could not be properly installed on the host on which the **ComponentInstallation** object resides.

remove

The **remove** operation removes a previously installed component package. The **implUUID** refers to the particular implementation within the component package. The operation raises an UnknownImplId exception if the **id** denoted by the **implUUID** parameter does not correspond to a previously installed component package. The operation raises a RemoveFailure exception if the installed component package that corresponds to the **id** denoted by the **implUUID** parameter could not be removed for any internal reasons.

get_implementation

The **get_implementation** operation returns the location of a component implementation. The **implUUID** refers to a particular implementation within a component package. The operation raises an UnknownImplId exception if the **id** denoted by the **implUUID** parameter does not correspond to a previously installed component package. The operation raises an InstallationFailure exception if a component package corresponding to the **id** denoted by the **implUUID** parameter does not exist or cannot be accessed in order to retrieve a component implementation.

6.9.3 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. At least one **AssemblyFactory** object must be present on a deployment domain.

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 69-548).

~~**exception InvalidLocation { };**~~
exception InvalidAssembly { };

Note Ð Issue 5577

```
interface AssemblyFactory {
    Cookie create_assembly(in Location assembly_loc)
raises (InvalidLocation, CreateFailure);
    Assembly lookup(in Cookie c)
raises (InvalidAssembly);
    void destroy(in Cookie c)
raises (InvalidAssembly,  RemoveFailure);
};
```

create_assembly

The **create_assembly** operation creates an **Assembly** object on the host on which the **AssemblyFactory** is located. It takes a string denoting the location of an assembly package and returns a **Cookie** that may be used to reference the assembly later on. The **Cookie** is the same as specified in Section 1.5.2.4, ÒCookie type,Ó on page 1-18 of this document. The operation raises an InvalidLocation exception if the assembly package could not be found. The operation raises a CreateFailure exception if an **Assembly** object could not be created for internal reasons such as insufficient resources.

lookup

The **lookup** operation takes a **Cookie** and returns an object reference to an **Assembly** object. The operation raises an InvalidAssembly exception if the **Cookie** does not refer to an existing assembly that was previously created by this **AssemblyFactory**.

destroy

The **destroy** operation destroys the assembly referenced by a **Cookie**. If the assembly is active it will first tear down the assembly. The operation raises an InvalidAssembly exception if the **Cookie** does not reference an existing assembly that was previously created by this **AssemblyFactory**. The operation raises a RemoveFailure exception if the **Assembly** was not properly and completely removed for any internal reasons.

6.9.4 Assembly Interface

The **Assembly** interface represents an assembly instantiation. It is used to build up and tear down component assemblies. Building the assembly up means instantiating all of the components in the assembly and creating connections between them as specified in the assembly descriptor. Tearing the assembly down means removing all connections and destroying the components, homes, and containers, and component servers created by the assembly.

```
enum AssemblyState { INACTIVE,  INSERVICE };

interface Assembly {
    void build() raises (CreateFailure);
    void tear_down() raises (RemoveFailure);
    AssemblyState get_state();
};
```

build

The **build** operation creates required component servers, creates required containers, installs required component homes, instantiates components, configures and interconnects them according to the assembly descriptor. If the build fails, then the

build operation is responsible for cleaning up any pieces of the assembly that were created. The operation raises a CreateFailure exception if an assembly could not be built for internal reasons such as insufficient resources.

tear_down

The **tear_down** operation removes all connections between components and destroys all components, homes, containers, and component servers that were created by the **build** operation. The operation raises a RemoveFailure exception if the assembly could not be removed properly and completely for any internal reasons.

get_state

The **get_state** operation returns whether the assembly is active or inactive. An assembly will be inactive before it is built, while it is being built, when it is being torn down, and after it has been torn down. It will be active after it is successfully built and before it is torn down.

6.9.5 ServerActivator interface

The **ServerActivator** interface represents a singleton object that resides on each host and acts as a factory for **ComponentServer** objects. A **ServerActivator** object is used by an **Assembly** to create a **ComponentServer** object that is then used by the **Assembly** to create **Container** objects.

```
typedef sequence<ComponentServer> ComponentServers;

interface ServerActivator {
    ComponentServer create_component_server(
                    in ConfigValues config)
                raises(CreateFailure, InvalidConfiguration);
    void remove_component_server(in ComponentServer server)
                raises(RemoveFailure);
    ComponentServers get_component_servers();
};
```

create_component_server

The **create_component_server** operation creates a **ComponentServer** object on the host on which the **ServerActivator** is located. The **config** parameter represents a sequence of **ConfigValue** objects that provide name value pairs used to configure the new **ComponentServer** such as the required ORB and programming language. The operation raises an InvalidConfiguration exception if the configuration values are not valid. The operation raises a CreateFailure exception if a component server could not be created for internal reasons such as insufficient resources.

remove_component_server

The **remove_component_server** operation removes a component server and all containers, homes, and components created in this component server. The **server** parameter points to the specifc component server that should be removed. The operation raises a BAD_PARAM system exception if the **server** parameter was not created by this **ServerActivator**. The operation raises a RemoveFailure exception if the component server or any of its containers could not be removed for any internal reasons.

get_component_servers

The **get_component_servers** operation returns a sequence of all **ComponentServer** objects created by this **ServerActivator**.

6.9.6 ComponentServer interface

The **ComponentServer** acts as a singleton factory for the creation of **Container** objects and is used by an **Assembly** object during the deployment process.

```
typedef sequence<Container> Containers;

interface ComponentServer {
    readonly attribute ConfigValues configuration;
    ServerActivator get_server_activator();
    Container create_container(in ConfigValues config)
                    raises (CreateFailure, InvalidConfiguration);
    void remove_container(in Container cref) raises (RemoveFailure);
    Containers get_containers();
    void remove() raises (RemoveFailure);
};
```

configuration

Accessing the **configuration** attribute returns the sequence of **ConfigValue** objects passed in to the **create_component_server** operation at the **ComponentServer** creation.

get_server_activator

The **get_server_activator** operation returns a reference to the **ServerActivator** object that has created this **ComponentServer**.

create_container

The **create_container** operation creates a new **Container** object within the component server process. The **config** parameter represents a sequence of **ConfigValue** objects that provide name value pairs used to configure the new

Container. The operation raises an InvalidConfiguration exception if the configuration values are not valid. The operation raises a CreateFailure exception if a container could not be created for internal reasons such as insufficient resources.

remove_container

The **remove_container** operation removes a previously created container and all homes and components that were created in the container. The **cref** parameter refers to the container that should be removed. The operation raises a BAD_PARAM system exception if the **cref** parameter was not created by this **ComponentServer**. The operation raises a RemoveFailure exception if the container or any of its homes could not be removed for any internal reasons.

get_containers

The **get_containers** operation returns a sequence of all **Container** objects created by this **ComponentServer**.

remove

The **remove** operation removes the component server and all containers, homes, and components that were created in the component server. The operation raises a RemoveFailure exception if the component server or any of its containers could not be removed for any internal reasons.

6.9.7 Container interface

The **Container** interface represents 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 multiple component homes and component instances. The following constraints apply:

¥ Each container has an associated **container API type**, which describes its interaction with the component, and an associated **CORBA usage model**, which describes its interaction with the POA, the ORB, and a set of CORBA services,

¥ Each container supports a single **container API type** and manages a specific **component category**. Multiple home instances of the same **component category** can be deployed in the same container.

```
module Components {
    typedef sequence<CCMHome> CCMHomes;
};

exception ImplEntryPointNotFound { };

interface Container {
    readonly attribute ConfigValues configuration;
    ComponentServer get_component_server();
    CCMHome install_home(in UUID id,
```

```
                                        in string entrypt,
                                        in ConfigValues config)
                        raises (UnknownImplId,
                                    ImplEntryPointNotFound,
                                    InstallationFailure,
                                    InvalidConfiguration);
        void remove_home(in CCMHome href) raises (RemoveFailure);
        CCMHomes get_homes();
        void remove() raises (RemoveFailure);
    };
```

configuration

Accessing the **configuration** attribute returns the sequence of **ConfigValue** objects passed in to the **create_container** operation at the **Container** creation.

get_component_server

The **get_component_server** operation returns a reference to the **ComponentServer** object that created this **Container**.

install_home

The **install_home** operation installs and returns a new **CCMHome** object. The **id** and **entrypt** parameters are used by the container to locate an implementation file and instantiate a new home object. The **config** parameter represents a sequence of **ConfigValue** objects that provide name value pairs used to configure the installation of the new home instance; for example, provide persistency source, transaction, and security policies that must be applied to the home and its components. The operation raises an UnknownImplId exception if the **id** parameter does not correspond to any component packages installed using the **ComponentInstallation** object. The operation raises an ImplEntryPointNotFound exception if the **entrypt** parameter cannot be found in the implementation returned from **ComponentInstallation**. The operation raises an InstallationFailure exception if the home could not be installed in the container for internal reasons such as insufficient resources or inadequate implementation for this container; for example, installing a C++ home implementation in a Java container. The operation raises an InvalidConfiguration exception if the **config** parameter does not contain valid configuration name value pairs.

remove_home

The **remove_home** operation removes a home from the container and all components that were created by this home. The operation raises a BAD_PARAM system exception if the **href** parameter was not created by this container. The operation raises a RemoveFailure exception if the home or any of its components could not be removed from the container for any internal reasons.

get_homes

The **get_homes** operation returns a sequence of all **CCMHome** objects created by this **Container**.

remove

The **remove** operation removes the container and all homes and components created in the container. The operation raises a RemoveFailure exception if the container or any of its homes could not be removed for any internal reasons.

6.9.8 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 componentÕs 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. This generic entry point function allows a container to create a component home without requiring 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++.

In general, the entry point function takes no arguments and returns a pointer or reference to a **HomeExecutorBase**.

Entry Points in Java

In Java, the entry point is the name of a class and static method that 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 Ò**bigbank.corbacomponents.Account.AccountHomeFactory.create**Ó.

Entry Points in C++

In C++, the entry point is the symbol in a shared library or DLL that should be invoked to return the **HomeExecutorBase** for the componentÕs home implementation.

The entry point should have ÒCÓ 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 Ò**createAccountHome**.Ó

XML DTDs

7

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

This chapter contains the definitions of the XML DTDs used by the CORBA Components.

Contents

This chapter contains the following sections.

Section Title	Page
Òsoftpkg.dtdÓ	7-1
Òcorbacomponent.dtdÓ	7-6
Òproperties.dtdÓ	7-12
Òcomponentassembly.dtdÓ	7-13

7.1 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?
       , usage?
     ) >
<!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
     ( softpkgref
     | codebase
     | fileinarchive
     | localfile
     | name
     | valuetypefactory
     ) >
<!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
     ) >
```

Note Ð Issue 5091

```
<!ATTLIST idl
     id CDATA #REQUIRED
     homeid CDATA #REQUIRED >

<!ELEMENT implementation
     ( description
     | code
     | compiler
     | dependency
     | descriptor
     | extension
     | programminglanguage
     | humanlanguage
     | os
     | propertyfile
     | processor
     | runtime
     )* >
<!ATTLIST implementation
     id ID #IMPLIED
     variation CDATA #IMPLIED >

<!ELEMENT implref EMPTY >
<!ATTLIST implref
     idref CDATA #REQUIRED >
```

Note Ð Issue 5584

```
<!ELEMENT ins EMPTY>
```

```
<!ATTLIST ins
     name CDATA #REQUIRED >

<!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 ) >
```

Note Ð Issue 5584

```
<!ELEMENT objref EMPTY>
<!ATTLIST objref
     string CDATA #REQUIRED >

<!ELEMENT os EMPTY >
<!ATTLIST os
     name    CDATA #REQUIRED
     version CDATA #IMPLIED>

<!ELEMENT pkgtype ( #PCDATA ) >
<!ATTLIST pkgtype
     version CDATA #IMPLIED >

<!ELEMENT processor EMPTY >
<!ATTLIST processor
     name CDATA #REQUIRED >

<!ELEMENT programminglanguage EMPTY>
<!ATTLIST programminglanguage
     name CDATA #REQUIRED
     version  CDATA #IMPLIED >

<!ELEMENT propertyfile
     ( fileinarchive
     | link) >
<!ATTLIST propertyfile
     type CDATA #IMPLIED >

<!ELEMENT repository
     ( ins
```

```
     | objref
     | link
     ) >
<!ATTLIST repository
     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 softpkgref
     ( ( fileinarchive
       | link
       )
       , implref?
     ) >

<!ELEMENT title ( #PCDATA ) >

<!ELEMENT usage ( #PCDATA ) >

<!ELEMENT valuetypefactory
     ( codebase
     | fileinarchive
     | link
     ) >
<!ATTLIST valuetypefactory
     repid             CDATA #REQUIRED
     valueentrypoint   CDATA #IMPLIED
     factoryentrypoint CDATA #IMPLIED >

<!ELEMENT webpage ( #PCDATA ) >
<!ATTLIST webpage
     %simple-link-attributes; >
```

7.2 corbacomponent.dtd

```
<!-- DTD for CORBA Component Descriptor. The root element is
     <corbacomponent>. 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 accessmode EMPTY>
<!ATTLIST accessmode
    mode (READ_ONLY|READ_WRITE) #REQUIRED >
```

Note Ð Issue 5499

~~<!ELEMENT catalog EMPTY>~~

~~<!ATTLIST catalog~~

~~type CDATA #REQUIRED >~~

```
<!ELEMENT componentfeatures
    ( inheritscomponent?
    , supportsinterface*
    , ports
    , operationpolicies?
    , extension*
    ) >
<!ATTLIST componentfeatures
    name  CDATA #REQUIRED
    repid CDATA #REQUIRED >

<!ELEMENT componentkind
     ( service
     | session
     | process
     | entity
     | unclassified
     ) >

<!ELEMENT componentproperties
     ( fileinarchive
     ) >

<!ELEMENT componentrepid EMPTY >
<!ATTLIST componentrepid
     repid CDATA #IMPLIED >
```

Note Ð Issue 5499

```
<!ELEMENT containermanagedpersistence
```

```
    ( storagehome
    , pssimplementation?
    ~~, catalog?~~
    , accessmode
    , psstransaction
    , params?
    ) >

<!ELEMENT configurationcomplete EMPTY >
<!ATTLIST configurationcomplete
     set ( true | false ) #REQUIRED >

<!ELEMENT consumes
     ( eventpolicy
     , extension* ) >
<!ATTLIST consumes
     consumesname CDATA #REQUIRED
     eventtype    CDATA #REQUIRED >

<!ELEMENT corbacomponent
     ( corbaversion
     , componentrepid
     , homerepid
     , componentkind
     , interop?
     , transaction?
     , security?
     , threading
     , configurationcomplete
     , extendedpoapolicy*
     , repository?
     , segment*
     , componentproperties?
     , homeproperties?
     , homefeatures+
     , componentfeatures+
     , interface*
     , extension*
     ) >

<!ELEMENT corbaversion (#PCDATA) >

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT emits
     ( eventpolicy
     , extension* ) >
<!ATTLIST emits
     emitsname CDATA #REQUIRED
     eventtype CDATA #REQUIRED >

<!ELEMENT entity
     ( servant ) >

<!ELEMENT eventpolicy EMPTY>
```

```
<!ATTLIST eventpolicy
     policy ( 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 >

<!-- 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 homefeatures
    ( inheritshome?
    , operationpolicies?
    , extension* ) >
<!ATTLIST homefeatures
    name  CDATA #REQUIRED
    repid CDATA #REQUIRED >

<!ELEMENT homeproperties
     ( fileinarchive
     ) >

<!ELEMENT homerepid EMPTY >
<!ATTLIST homerepid
     repid CDATA #IMPLIED >

<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
    repid CDATA #REQUIRED>

<!ELEMENT inheritshome EMPTY>
<!ATTLIST inheritshome
    repid CDATA #REQUIRED>

<!ELEMENT inheritsinterface EMPTY>
<!ATTLIST inheritsinterface
    repid CDATA #REQUIRED>
```

```
<!ELEMENT ins EMPTY>
<!ATTLIST ins
     name CDATA #REQUIRED >

<!ELEMENT interface
     ( inheritsinterface*
     , operationpolicies? ) >
<!ATTLIST interface
    name  CDATA #REQUIRED
    repid CDATA #REQUIRED >

<!ELEMENT interop EMPTY>
<!ATTLIST interop
     type CDATA #REQUIRED
     direction ( hasview | isview ) #REQUIRED
     descriptor CDATA  #REQUIRED >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
     %simple-link-attributes; >

<!ELEMENT objref EMPTY>
<!ATTLIST objref
     string CDATA #REQUIRED >

<!ELEMENT operation
     ( transaction?
     , requiredrights? ) >
<!ATTLIST operation
     name CDATA #REQUIRED >
<!-- an operation name of "*" specifies all operations in the current
scope -->

<!ELEMENT operationpolicies
     ( operation+ ) >

<!ELEMENT param EMPTY >
<!ATTLIST param
     name CDATA #REQUIRED
     value CDATA #REQUIRED >

<!ELEMENT params (param+) >

<!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
     | publishes
     | consumes
     )* >

<!ELEMENT process
     ( servant ) >

<!ELEMENT provides
     ( operationpolicies?
     , extension* ) >
<!ATTLIST provides
     providesname CDATA #REQUIRED
     repid        CDATA #REQUIRED
     facettag     CDATA #REQUIRED >

<!ELEMENT pssimplementation EMPTY>
<!ATTLIST pssimplementation
    id CDATA #REQUIRED >

<!ELEMENT psstransaction (psstransactionisolationlevel?) >
<!ATTLIST psstransaction
    policy (TRANSACTIONAL|NON_TRANSACTIONAL) #REQUIRED >

<!ELEMENT psstransactionisolationlevel EMPTY>
<!ATTLIST psstransactionisolationlevel
   level (READ_UNCOMMITTED|READ_COMMITTED|REPEATABLE_READ|SERIALIZABLE)
#REQUIRED >

<!ELEMENT publishes
     ( eventpolicy
     , extension* ) >
<!ATTLIST publishes
     publishesname CDATA #REQUIRED
     eventtype CDATA #REQUIRED >

<!ELEMENT repository
     ( ins
     | objref
     | link
     ) >
<!ATTLIST repository
     type CDATA #IMPLIED >

<!ELEMENT requiredrights
     ( right* ) >

<!ELEMENT right
     ( description? ) >
<!ATTLIST right
     name CDATA #REQUIRED >
```

```
<!ELEMENT security
     ( requiredrights? ) >
<!ATTLIST security
     rightsfamily CDATA #REQUIRED
     rightscombinator (secallrights | secanyrights) #REQUIRED >

<!ELEMENT segment
    ( segmentmember+
    , containermanagedpersistence?
    , extension*
    ) >
<!ATTLIST segment
    name       CDATA #REQUIRED
    segmenttag CDATA #REQUIRED >

<!ELEMENT segmentmember EMPTY>
<!ATTLIST segmentmember
     facettag CDATA #REQUIRED >

<!ELEMENT servant EMPTY >
<!ATTLIST servant
     lifetime (component|method|transaction|container) #REQUIRED >

<!ELEMENT service EMPTY >

<!ELEMENT session
     ( servant ) >

<!ELEMENT storagehome EMPTY>
<!ATTLIST storagehome
    id CDATA #REQUIRED >

<!ELEMENT supportsinterface
     ( operationpolicies?
     , extension* ) >
<!ATTLIST supportsinterface
     repid CDATA #REQUIRED >

<!ELEMENT threading EMPTY>
<!ATTLIST threading
     policy ( serialize | multithread ) #REQUIRED >

<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
     use (self-managed|not-supported|required|supports|requires-
new|mandatory|never) #REQUIRED >

<!ELEMENT unclassified
     ( poapolicies ) >

<!ELEMENT uses ( extension* ) >
<!ATTLIST uses
     usesname CDATA #REQUIRED
     repid    CDATA #REQUIRED >
```

7.3 properties.dtd

```
<!-- DTD for CORBA Component property file. The root element
     is <properties>. Elements are listed alphabetically.
  -->

<!ELEMENT choice ( #PCDATA ) >

<!ELEMENT choices ( choice | range )+ >

<!ELEMENT defaultvalue ( #PCDATA ) >

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT value ( #PCDATA ) >

<!ELEMENT properties
     ( description?
     , ( simple
       | sequence
       | struct
       | valuetype
       )*
     ) >

<!ELEMENT range (value, value) >

<!ELEMENT simple
     ( description?
     , value
     , choices?
     , defaultvalue?
     ) >
```

Note Ð Issue 5429

```
<!ATTLIST simple
     name CDATA #IMPLIED
     type ( boolean
          | char
          | double
          | float
          | short
          | long
          | objref
          | octet
```
~~| short~~
```
          | string
          | ulong
          | ushort
          | longlong
          | ulonglong
          | longdouble
          | wchar
          | wstring
```

```
          | fixed
) #REQUIRED >

<!ELEMENT sequence
     ( description?
     , ( simple*
       | struct*
       | sequence*
       | valuetype*
       )
     ) >
<!ATTLIST sequence
     name CDATA #IMPLIED
     type CDATA #REQUIRED >

<!ELEMENT struct
     ( description?
     , ( simple
       | sequence
       | struct
       | valuetype
       )*
     ) >
<!ATTLIST struct
     name CDATA #IMPLIED
     type CDATA #REQUIRED >

<!ELEMENT valuetype
     ( description?
     , ( simple
       | sequence
       | struct
       | valuetype
       )*
     ) >
<!ATTLIST valuetype
     name CDATA #IMPLIED
     type CDATA #REQUIRED
     primarykey (true | false) "false" >
```

7.4 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
     ( description?
     , componentfiles
     , partitioning
     , connections?
     , extension*
     ) >
<!ATTLIST componentassembly
     id ID #REQUIRED
     derivedfrom CDATA #IMPLIED >

<!ELEMENT componentfile
     ( fileinarchive
     | codebase
     | link
     ) >
<!ATTLIST componentfile
     id   ID    #REQUIRED
     type CDATA #IMPLIED >

<!ELEMENT componentfileref EMPTY >
<!ATTLIST componentfileref
     idref CDATA #REQUIRED >

<!ELEMENT componentfiles
     ( componentfile+
     ) >

<!ELEMENT componentimplref EMPTY >
<!ATTLIST componentimplref
     idref CDATA #REQUIRED >

<!ELEMENT componentinstantiation
     ( usagename?
     , componentproperties?
     , registercomponent*
     , extension*
     ) >
<!ATTLIST componentinstantiation
     id          ID    #REQUIRED >

<!ELEMENT componentinstantiationref EMPTY >
<!ATTLIST componentinstantiationref
     idref CDATA #REQUIRED >

<!ELEMENT componentproperties
     ( fileinarchive
     | codebase
     ) >
```

```
<!ELEMENT componentsupportedinterface
     ( componentinstantiationref
     | findby
     )>
```

Note Ð Issue 5093

```
<!ELEMENT connectevent
     ( ( consumesport
       | existinginterface
       )
     , ( emitsport
       | publishesport
       )
     ) >
<!ATTLIST connectevent
     id ID #IMPLIED >

<!ELEMENT connecthomes
     ( proxyhome
     , destinationhome
     ) >
<!ATTLIST connecthomes
     id ID #IMPLIED >

<!ELEMENT connectinterface
     ( usesport
     , ( providesport
       | componentsupportedinterface
       | existinginterface
       | homeinterface
       )
     ) >
<!ATTLIST connectinterface
     id ID #IMPLIED >

<!ELEMENT connections
     ( connectinterface
     | connectevent
     | connecthomes
     | extension
     )* >

<!ELEMENT consumesidentifier ( #PCDATA ) >

<!ELEMENT consumesport
     ( consumesidentifier
     , ( componentinstantiationref
       | findby
       )
     )>

<!ELEMENT description ( #PCDATA ) >
```

```
<!ELEMENT destination ( #PCDATA ) >

<!ELEMENT destinationhome
     ( homeplacementref
     | findby
     ) >

<!ELEMENT emitsidentifier ( #PCDATA ) >

<!ELEMENT emitsport
     ( emitsidentifier
     , ( componentinstantiationref
       | findby
       )
     )>

<!ELEMENT executableplacement
     ( usagename?
     , componentfileref
     , componentimplref?
     , invocation?
     , destination?
     , extension*
     ) >
<!ATTLIST executableplacement
     id          ID    #REQUIRED
     cardinality CDATA "1" >

<!ELEMENT existinginterface
     ( findby )>

<!-- 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 independent of an 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 findby
     ( namingservice
```

```
     | stringifiedobjectref
     | traderquery
     | homefinder
     | extension
     ) >

<!ELEMENT homefinder EMPTY >
<!ATTLIST homefinder
     name CDATA #REQUIRED >

<!ELEMENT homeinterface
     ( homeplacementref
     | findby
     )>

<!ELEMENT homeplacement
     ( usagename?
     , componentfileref
     , componentimplref?
     , homeproperties?
     , componentproperties?
     , registerwithhomefinder*
     , registerwithnaming*
     , registerwithtrader*
     , componentinstantiation*
     , destination?
     , extension*
     ) >
<!ATTLIST homeplacement
     id          ID    #REQUIRED
     cardinality CDATA "1" >

<!ELEMENT homeplacementref EMPTY >
<!ATTLIST homeplacementref
     idref CDATA #REQUIRED >

<!ELEMENT homeproperties
     ( fileinarchive
     | codebase
     ) >

<!ELEMENT hostcollocation
     ( usagename?
     , impltype?
     , ( homeplacement
       | executableplacement
       | processcollocation
       | extension
       )+
     , destination?
     ) >
<!ATTLIST hostcollocation
     id          ID    #IMPLIED
     cardinality CDATA "1" >
```

```
<!ELEMENT impltype EMPTY >
<!ATTLIST impltype
     language CDATA #REQUIRED
     version  CDATA #IMPLIED >

<!ELEMENT invocation EMPTY >
<!ATTLIST invocation
     args CDATA    #REQUIRED >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
     %simple-link-attributes; >

<!ELEMENT namingservice EMPTY >
<!ATTLIST namingservice
     name CDATA #REQUIRED >

<!ELEMENT partitioning
     ( homeplacement
     | executableplacement
     | processcollocation
     | hostcollocation
     | extension
     )* >

<!ELEMENT processcollocation
     ( usagename?
     , impltype?
     , ( homeplacement
       | extension
       )+
     , destination?
     ) >
<!ATTLIST processcollocation
     id          ID    #IMPLIED
     cardinality CDATA "1" >

<!ELEMENT providesidentifier ( #PCDATA ) >

<!ELEMENT providesport
     ( providesidentifier
     , ( componentinstantiationref
       | findby
       )
     )>
```

Note Ð Issue 5497

```
<!ELEMENT proxyhome
     ( homeplacementref
     | findby
     ) >

<!ELEMENT publishesidentifier ( #PCDATA ) >
```

```
<!ELEMENT publishesport
     ( publishesidentifier
     , ( componentinstantiationref
       | findby
       )
     )>
```

Note Đ Issue 5092

```
<!ELEMENT registercomponent
     ( ( ~~emitsidentifier~~
       ~~|~~ providesidentifier
       | ~~publishes~~consumesidentifier
       )?
     , ( registerwithnaming
       | registerwithtrader
       )+
     ) >

<!ELEMENT registerwithhomefinder EMPTY >
<!ATTLIST registerwithhomefinder
     name CDATA #REQUIRED >

<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming
     name CDATA #IMPLIED >

<!ELEMENT registerwithtrader
     ( trader~~properties~~export ) >
<!ATTLIST registerwithtrader
     tradername CDATA #IMPLIED >
<!-- DEVNOTE: is tradername necessary? -->
<!-- DEVNOTE: Should trader properties be specified in component file?
And in assembly file? -->
```

Note Đ Issue 5497

```
~~<!ELEMENT proxyhome~~
~~     ( homeplacementref~~
~~     | findby~~
~~     ) >~~

<!ELEMENT stringifiedobjectref ( #PCDATA ) >

<!ELEMENT traderconstraint ( #PCDATA ) >

<!ELEMENT traderexport
     ( traderservicetypename
     , traderproperties
     ) >

<!ELEMENT traderpolicy
```

```
     ( traderpolicyname
     , traderpolicyvalue
     ) >

<!ELEMENT traderpolicyname ( #PCDATA ) >

<!ELEMENT traderpolicyvalue ( #PCDATA ) >

<!ELEMENT traderpreference ( #PCDATA ) >

<!ELEMENT traderproperties
     ( traderproperty+ ) >

<!ELEMENT traderproperty
     ( traderpropertyname
     , traderpropertyvalue
     ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ( #PCDATA ) >

<!ELEMENT traderquery
     ( traderservicetypename
     , traderconstraint
     , traderpreference?
     , traderpolicy*
     , traderspecifiedprop*
     ) >

<!ELEMENT traderservicetypename ( #PCDATA ) >

<!ELEMENT traderspecifiedprop ( #PCDATA ) >

<!ELEMENT usagename ( #PCDATA ) >

<!ELEMENT usesidentifier ( #PCDATA ) >

<!ELEMENT usesport
     ( usesidentifier
     , ( componentinstantiationref
       | findby
       )
     ) >
```

Interface Repository Metamodel 8

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

Contents

This chapter contains the following sections.

Section Title	Page
ÒIntroductionÓ	8-1
ÒConformance CriteriaÓ	8-26
ÒMOF DTDs and IDL for the Interface Repository MetamodelÓ	8-27

8.1 Introduction

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 8-1), the *ComponentIDL* Package is dependent upon the *BaseIDL* Package.

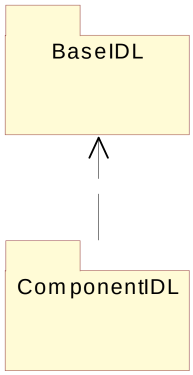


Figure 8-1 The Two Packages for the IDL Metamodel

8.1.1 BaseIDL Package

The base CORBA Interface Repository (IR) is described in the *Common Object Request Broker Architecture (CORBA)* 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; for example:

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.

8.1.1.1 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,Ó 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 IRÕs *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 specification 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 8-2 shows all of the metaclasses and relationships defined in the *BaseIDL* Package.

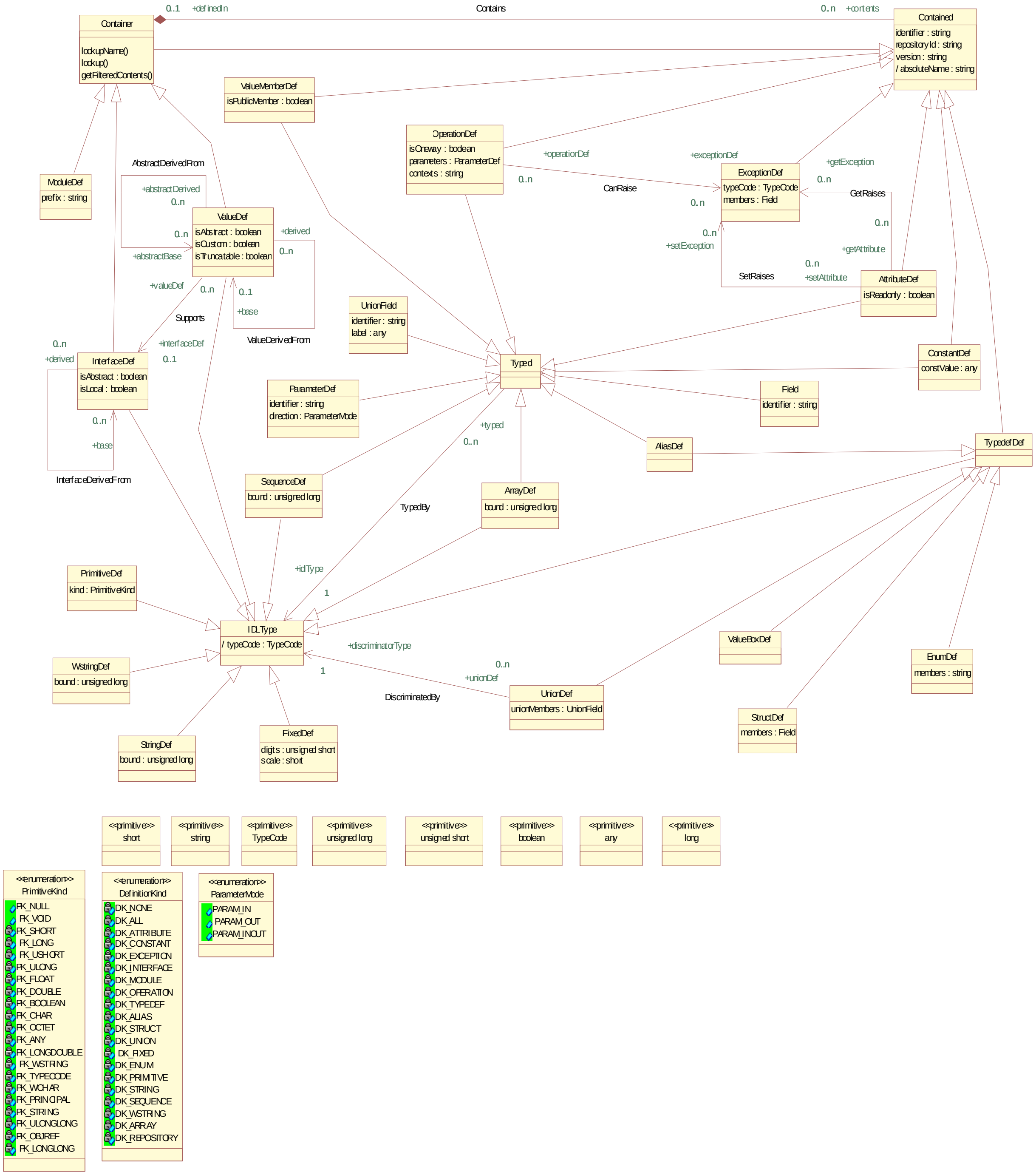


Figure 8-2 BaseIDL Package--All Elements

8.1.1.2 *Typing*

As mentioned earlier in this chapter (Section 8.1.1.1, ÒA Structural Comparison of the BaseIDL Package with the Existing IR,Ó on page 8-2), 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*.

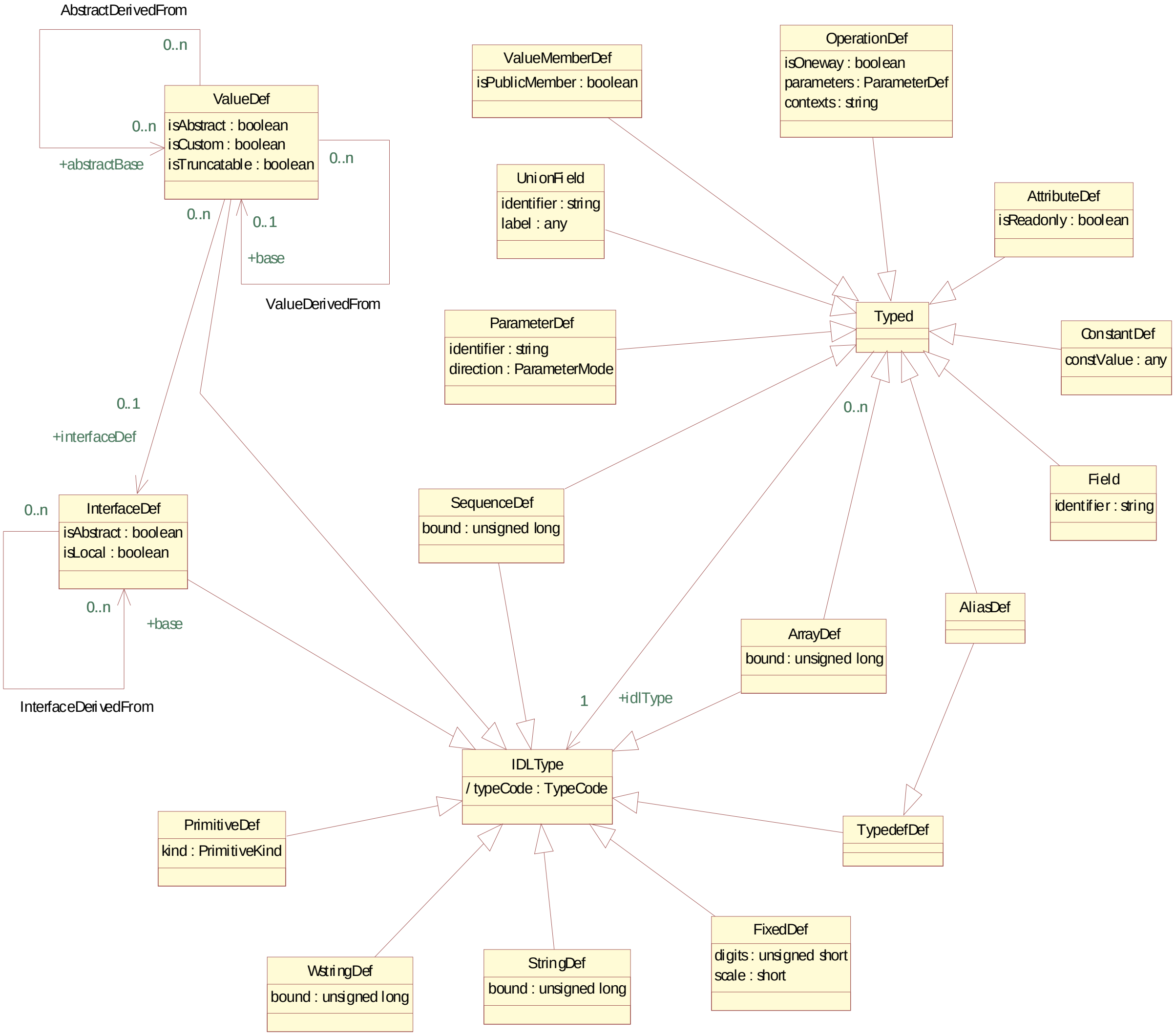


Figure 8-3 IDL Typing

8.1.1.3 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 8-4 expresses the containment hierarchy.

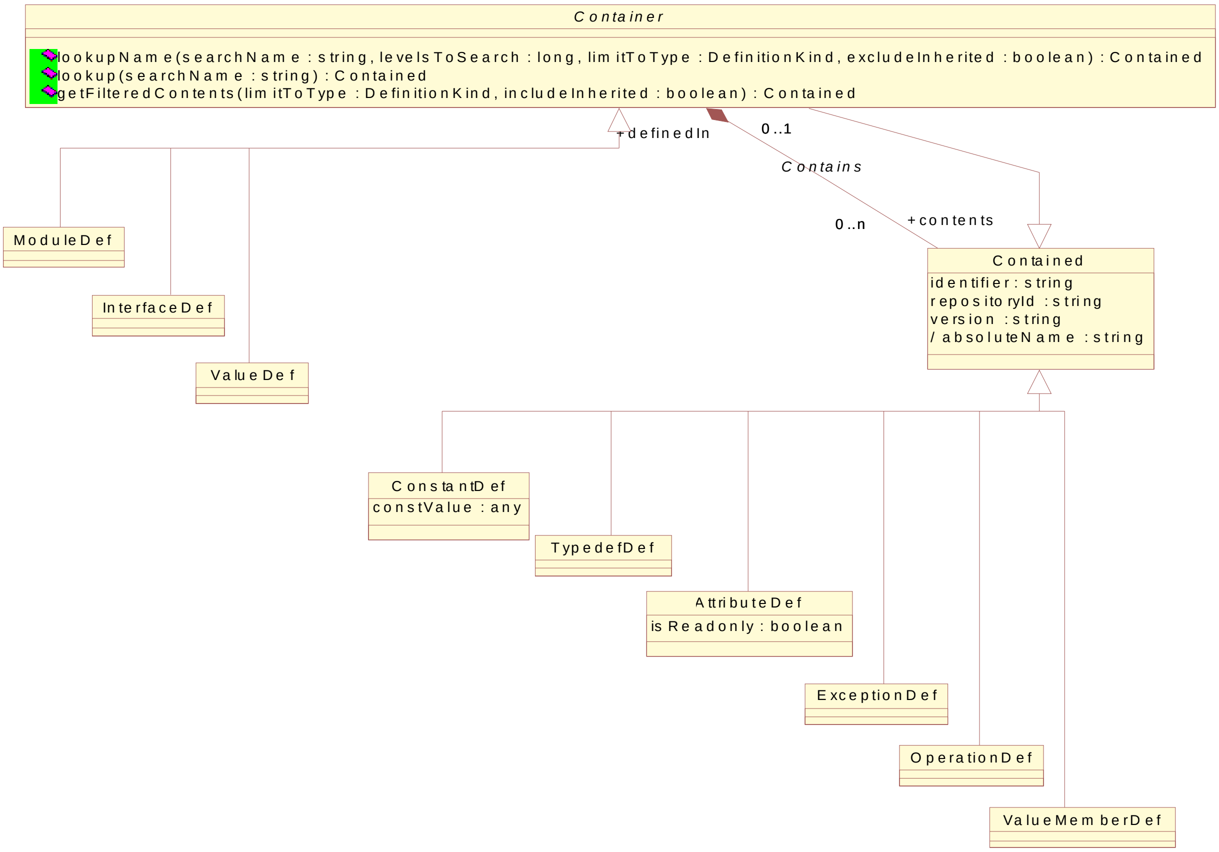


Figure 8-4 Containment Hierarchy

8.1.1.4 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 8-5 and Figure 8-6 express the containment constraints formally via the OMGÕs Object Constraint Language (OCL). They also supplement the formal expressions with English natural language equivalents.

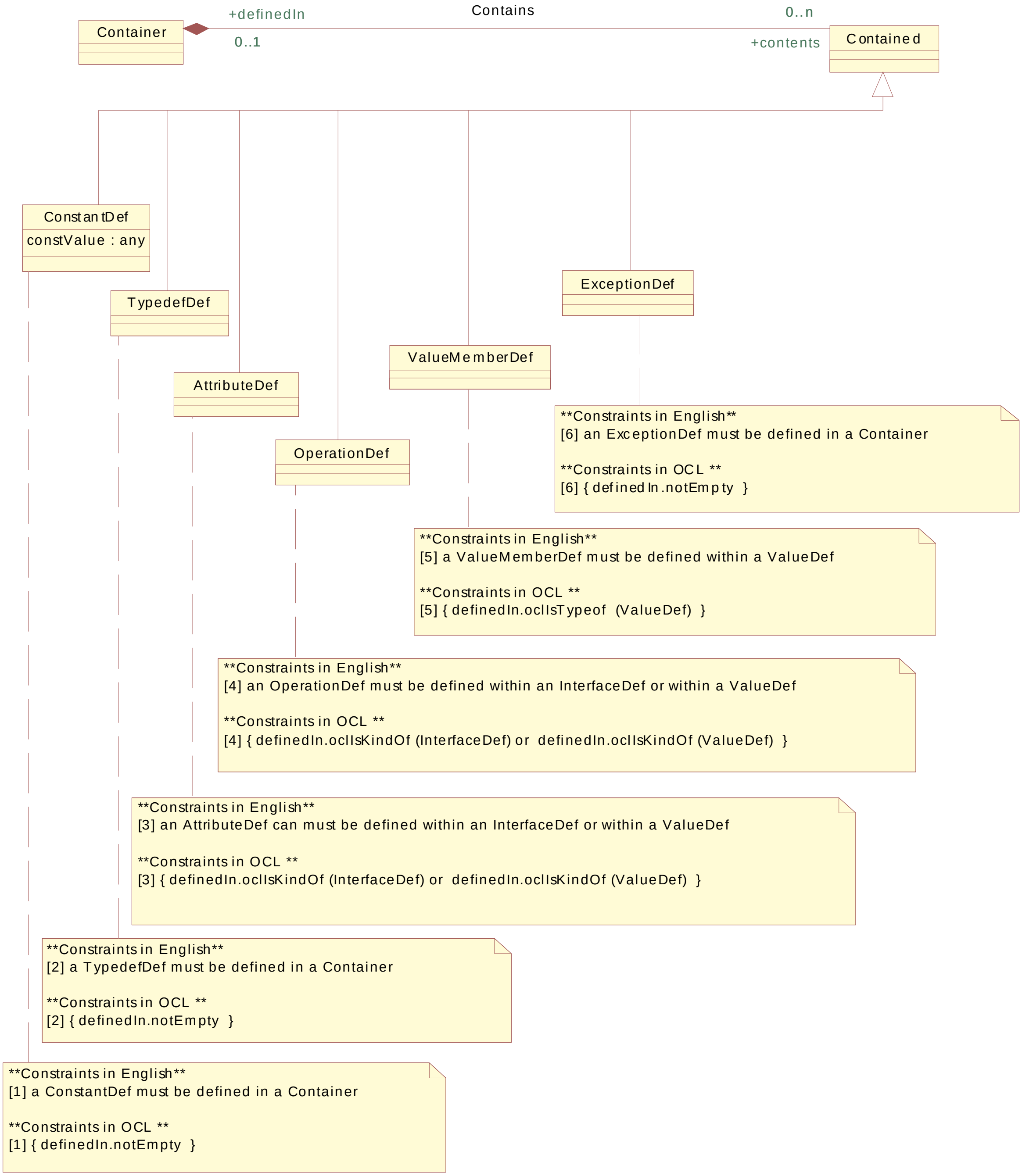


Figure 8-5 Containment Constraints--Subclasses of *Contained*

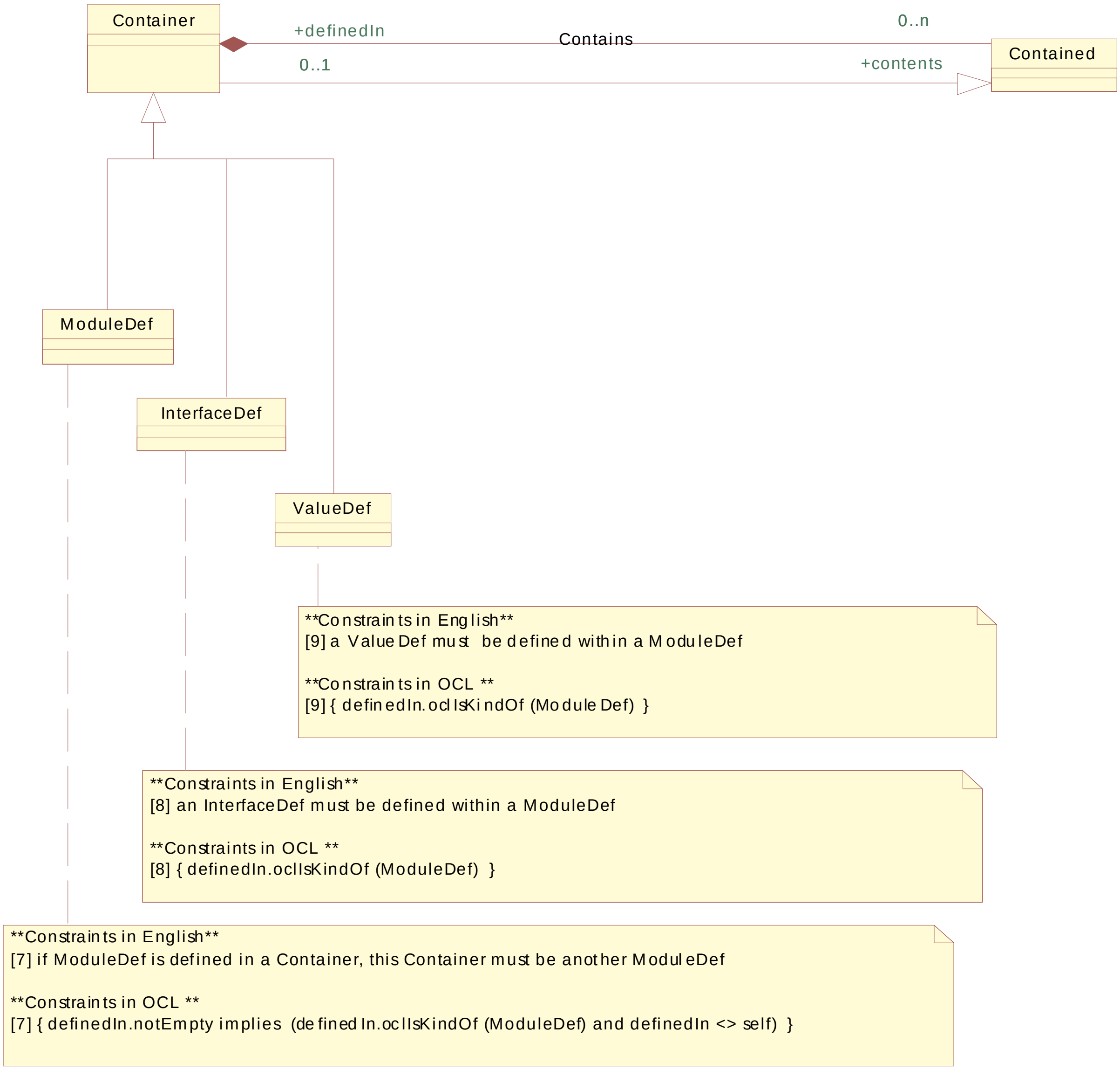


Figure 8-6 Containment Constraints--Subclasses of *Container*

8.1.1.5 *Typedef and Type Derivations*

Figure 8-7 expresses the hierarchy of derivatives of *Typedef* and *Typed*.

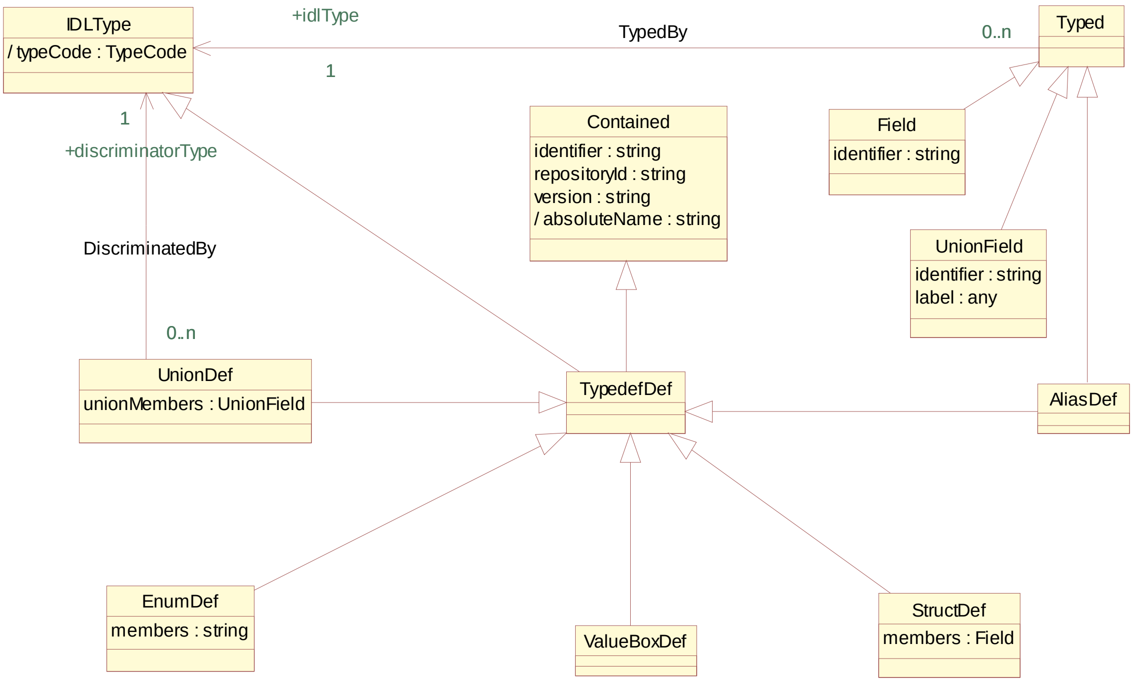


Figure 8-7 Derivations from *Typedef* and *Type*

8.1.1.6 Exceptions

Figure 8-8 shows the formal definition of the *ExceptionDef* metaclass. Note the inclusion of the newly-defined (in this specification) ability for attribute accessors and mutators to raise user-defined exceptions.

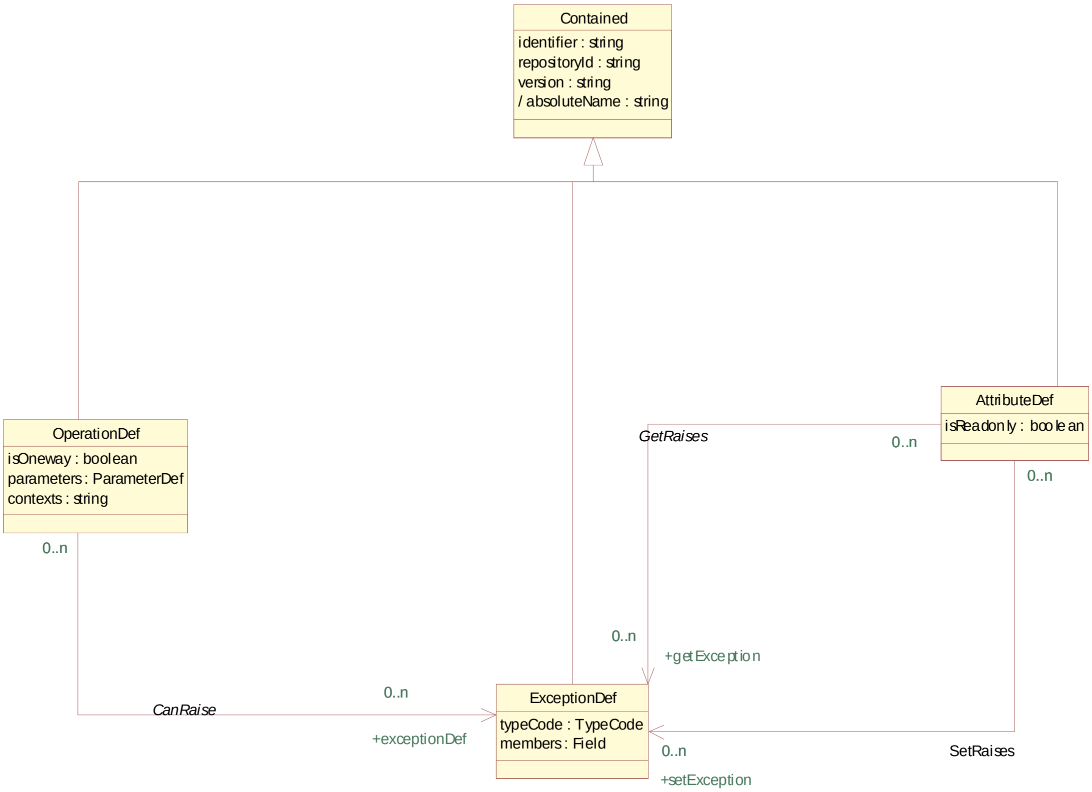
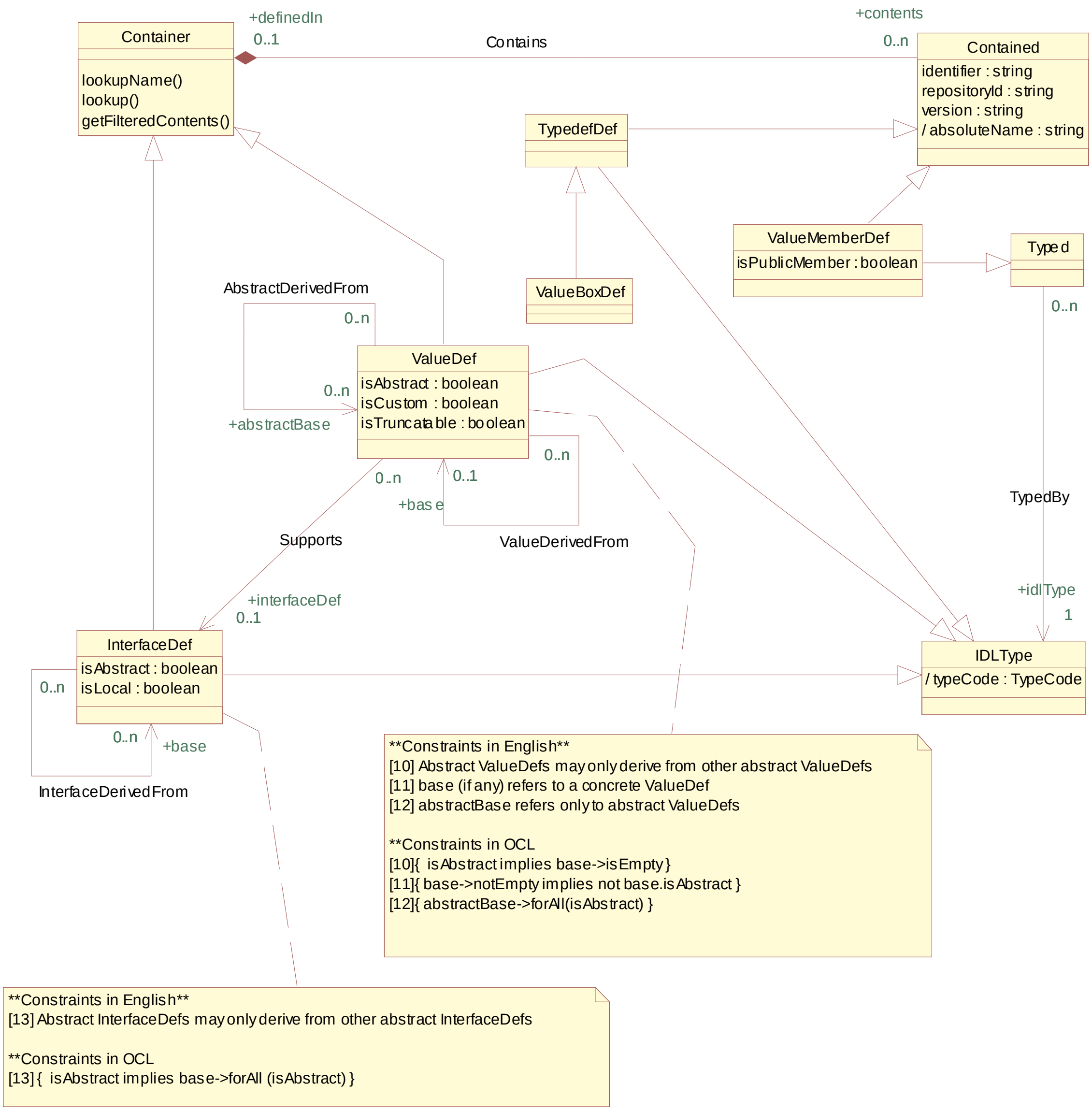


Figure 8-8 Exceptions

8.1.1.7 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 8-9 focuses on the definition of value types in the MOF-compliant IR metamodel.



Constraints in English
[10] Abstract ValueDefs may only derive from other abstract ValueDefs
[11] base (if any) refers to a concrete ValueDef
[12] abstractBase refers only to abstract ValueDefs

**Constraints in OCL
[10]{ isAbstract implies base->isEmpty}
[11]{ base->notEmpty implies not base.isAbstract }
[12]{ abstractBase->forAll(isAbstract) }

Constraints in English
[13] Abstract InterfaceDefs may only derive from other abstract InterfaceDefs

**Constraints in OCL
[13]{ isAbstract implies base->forAll (isAbstract) }

Figure 8-9 Value Types

8.1.1.8 Naming

Figure 8-10 focuses on the aspects of the metamodel that concern naming.

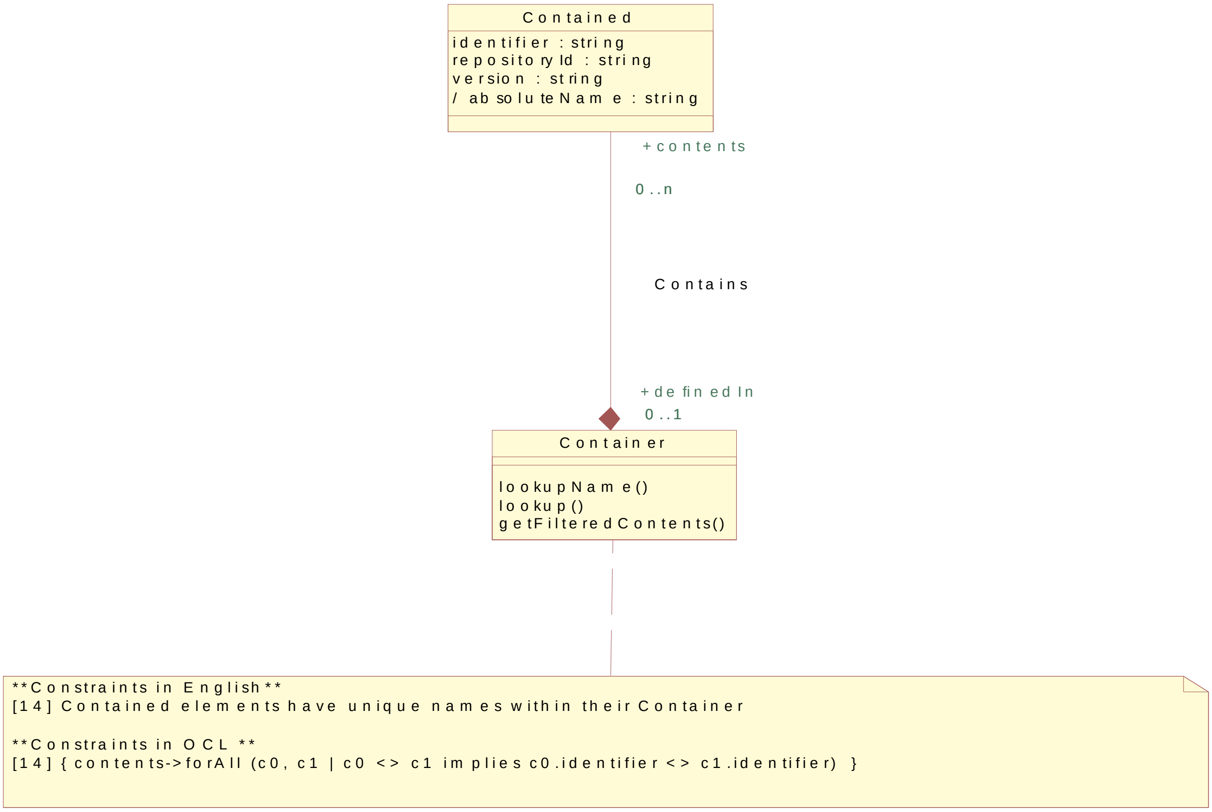


Figure 8-10 Naming

8.1.1.9 Operations

As mentioned earlier in this chapter (Section 8.1.1.1, ÒA Structural Comparison of the BaseIDL Package with the Existing IR,Ó on page 8-2), 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 8-11.

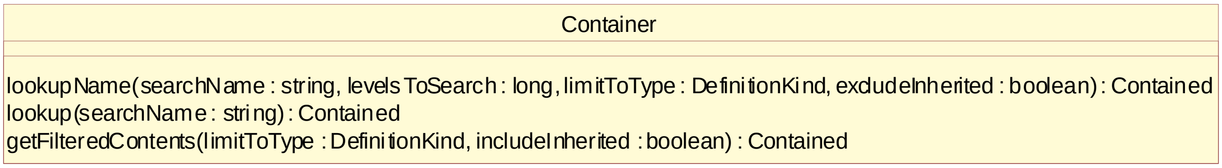


Figure 8-11 Convenience Operations

8.1.2 ComponentIDL Package

8.1.2.1 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.

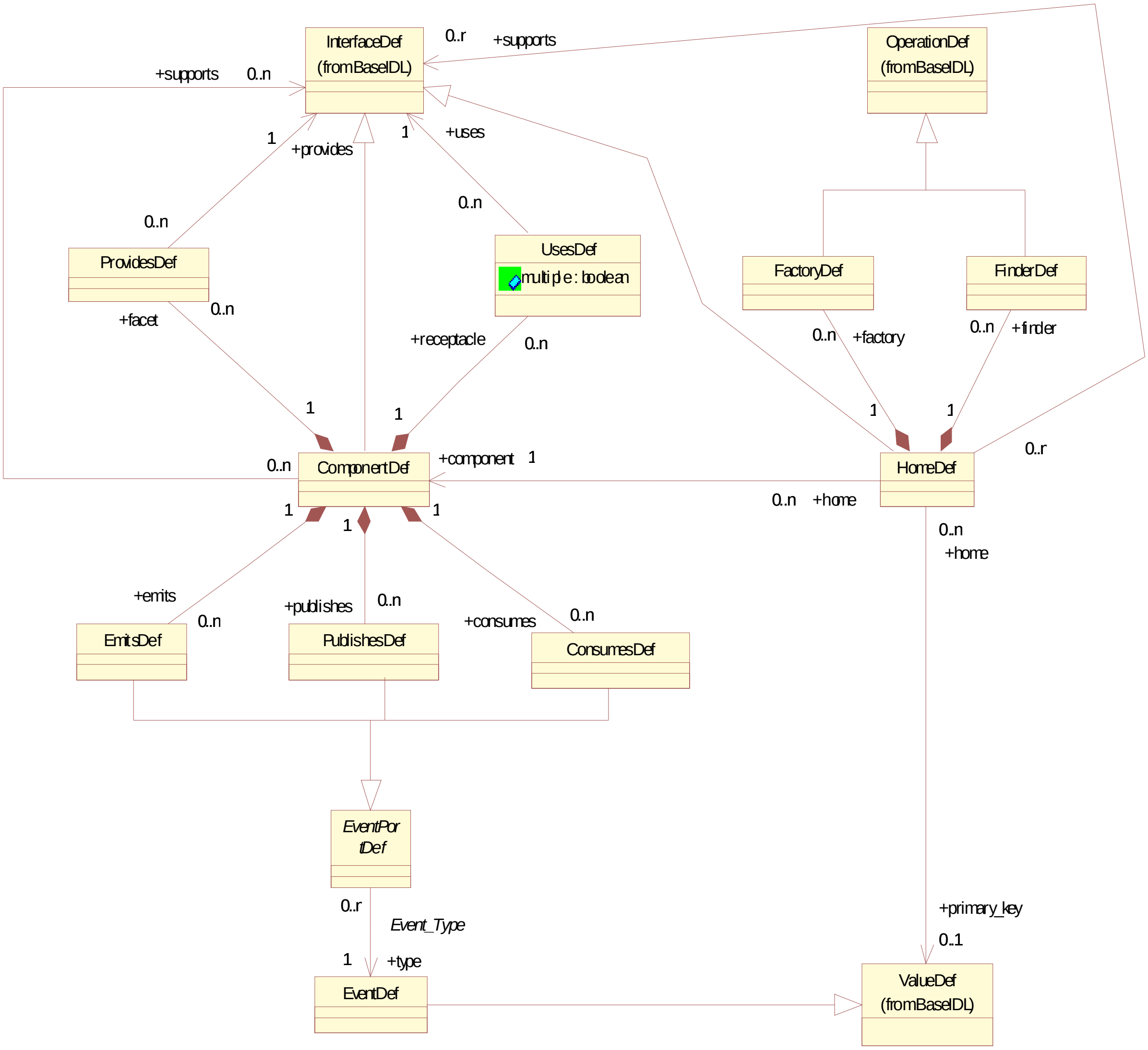


Figure 8-12 ComponentIDL Package - Main Diagram

8.1.2.2 Containers and Contained Elements

The following UML class diagram (Figure 8-13) describes the derivation of the metamodel elements from the BaseIDL *Container* and *Contained* elements:

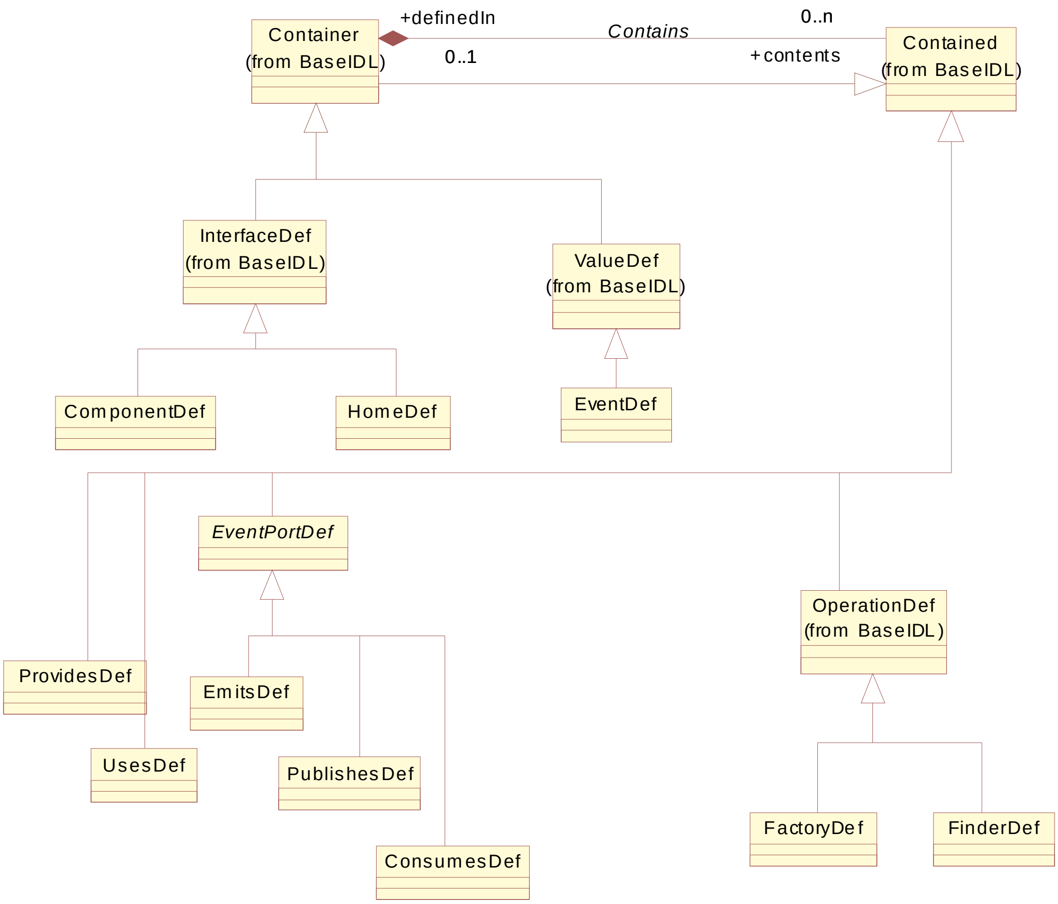
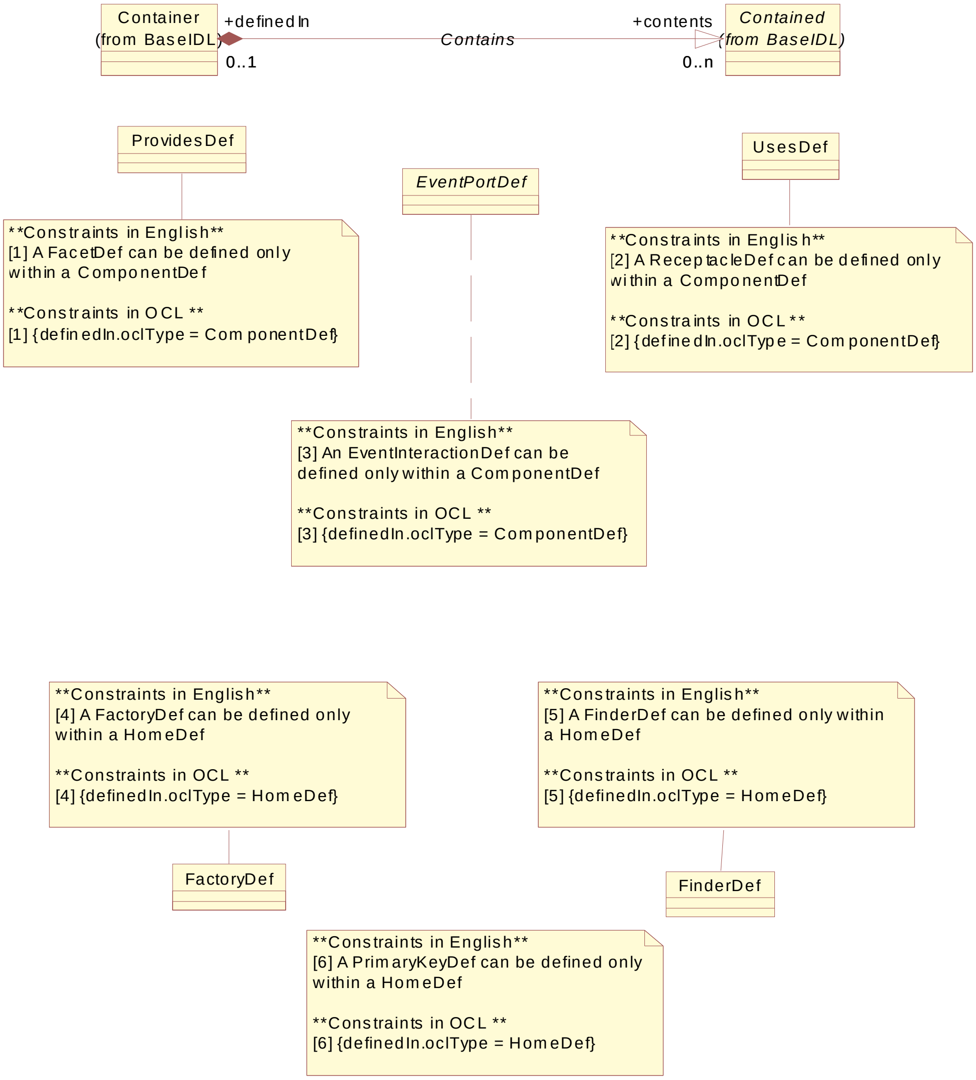


Figure 8-13 Containment Hierarchy

Each of the subtypes of *Contained* shown in Figure 8-13 can only be defined within certain subtypes of *Container*. Figure 8-14 formally specifies these constraints via the OMGÕs 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.



Constraints in English
[1] A FacetDef can be defined only within a ComponentDef

**Constraints in OCL **
[1] {definedIn.oclType = ComponentDef}

Constraints in English
[2] A ReceptacleDef can be defined only within a ComponentDef

**Constraints in OCL **
[2] {definedIn.oclType = ComponentDef}

Constraints in English
[3] An EventInteractionDef can be defined only within a ComponentDef

**Constraints in OCL **
[3] {definedIn.oclType = ComponentDef}

Constraints in English
[4] A FactoryDef can be defined only within a HomeDef

**Constraints in OCL **
[4] {definedIn.oclType = HomeDef}

Constraints in English
[5] A FinderDef can be defined only within a HomeDef

**Constraints in OCL **
[5] {definedIn.oclType = HomeDef}

Constraints in English
[6] A PrimaryKeyDef can be defined only within a HomeDef

**Constraints in OCL **
[6] {definedIn.oclType = HomeDef}

Figure 8-14 Constraints on Containment of Elements Defined In ComponentDef

An instance of *ComponentDef* describes a CORBA component in an abstract manner. The definition contains a description of all features of a component that are visible from the outside. More precise, it defines all interfaces including interfaces that are implicit or used for event communication. In detail, the features of component that are visible to the outside are:

- The component equivalent interface, containing all implicit operations, operations and attributes that are inherited by a component (also from supported interfaces), and attributes defined inside the component.

- The facets of a component; that is, all interfaces that are provided by the component to the outside.
- The receptacles of a component; that is, all interfaces that are used by a component.
- The events, which a component can emit, publish, or consume.

If a component is going to be implemented, all these features must be handled by the component implementation. To provide a common basis for defining the related implementation definitions (as part of CIF) the abstract metaclass *ComponentFeature* is defined. The metaclasses *ComponentDef*, *ProvidesDef*, *UsesDef*, and *EventPortDef* are defined as subclasses of the metaclass *ComponentFeature*.

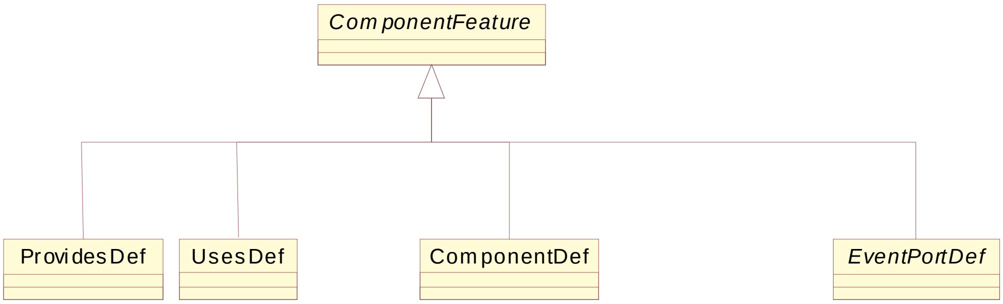


Figure 8-15

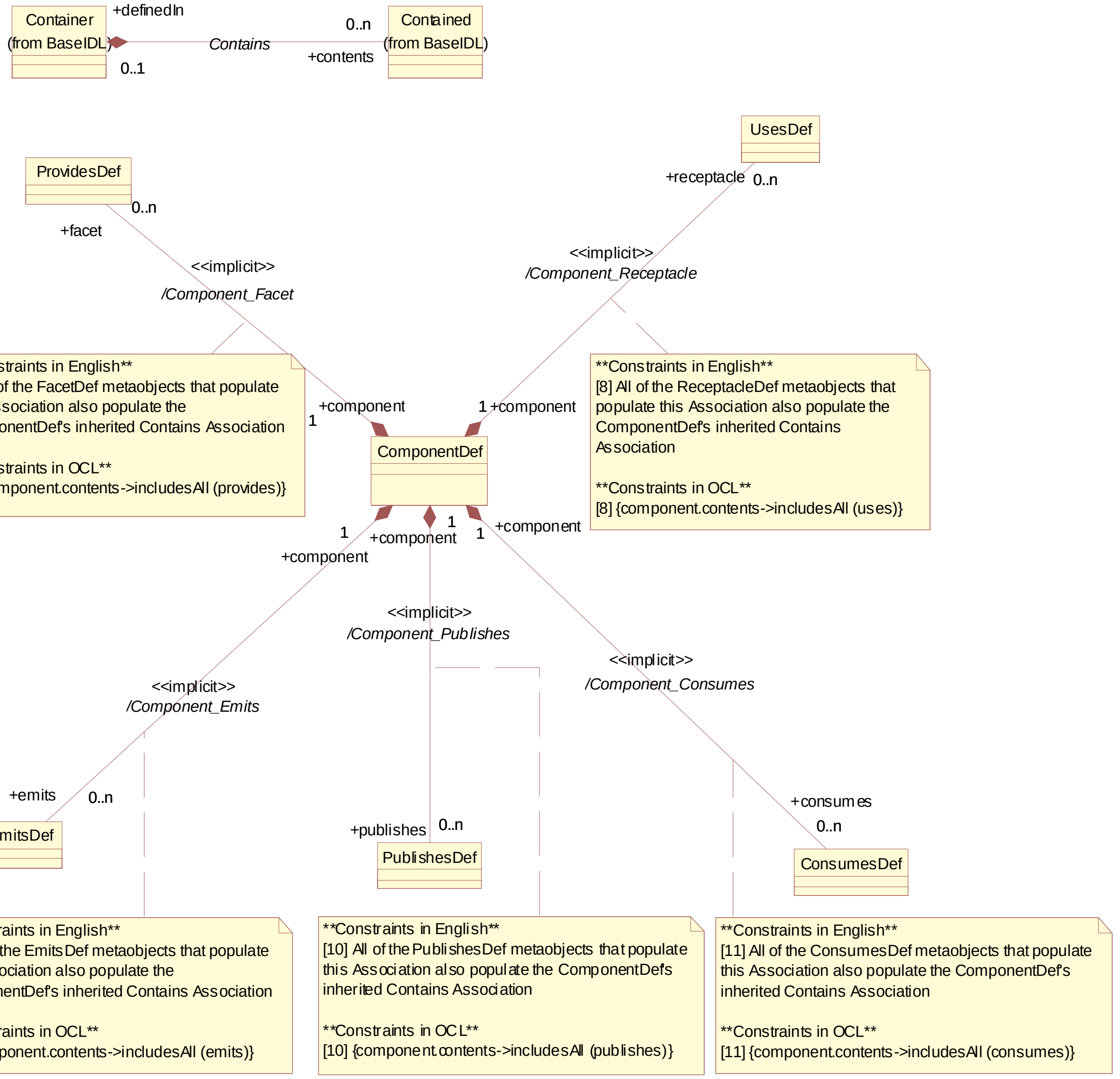
All of *ComponentDef*'s composition Associations shown in the main diagram (Figure 8-12 on page 8-15) are derived from the *BaseIDL* metamodelÕs *Contains* Association between *Container* and *Contained*. As shown by Figure 8-13 on page 8-16, *ComponentDef* inherits that Association from *InterfaceDef*, which inherits it from *Container.*

The following class diagram (Figure 8-16 on page 8-20) details these derived Associations. A Ò/Ó prefix in an Association name denotes that the Association is derived, and sets the MOF's ÒisDerivedÓ 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 ÒreflectiveÓ 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 Section 8.3, ÒMOF DTDs and IDL for the Interface Repository Metamodel,Ó on page 8-27.



Constraints in English
[7] All of the FacetDef metaobjects that populate this Association also populate the ComponentDef's inherited Contains Association

Constraints in OCL
[7] {component.contents->includesAll (provides)}

Constraints in English
[8] All of the ReceptacleDef metaobjects that populate this Association also populate the ComponentDef's inherited Contains Association

Constraints in OCL
[8] {component.contents->includesAll (uses)}

Constraints in English
[9] All of the EmitsDef metaobjects that populate this Association also populate the ComponentDef's inherited Contains Association

Constraints in OCL
[9] {component.contents->includesAll (emits)}

Constraints in English
[10] All of the PublishesDef metaobjects that populate this Association also populate the ComponentDef's inherited Contains Association

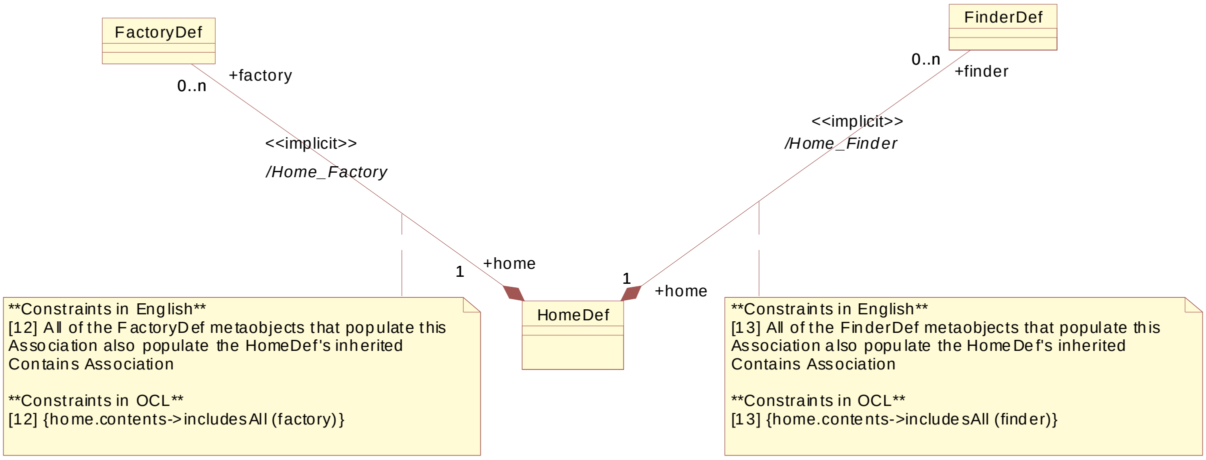
Constraints in OCL
[10] {component.contents->includesAll (publishes)}

Constraints in English
[11] All of the ConsumesDef metaobjects that populate this Association also populate the ComponentDef's inherited Contains Association

Constraints in OCL
[11] {component.contents->includesAll (consumes)}

Figure 8-16 Implicit Derived Containments with ComponentDef as the Composite

*HomeDef*Õs composition Associations also are derived from the *Contains* Association. As shown in Figure 8-13 on page 8-16, *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 8-17 formally defines the constraints that define the semantics of the derivations.



Constraints in English
[12] All of the FactoryDef metaobjects that populate this Association also populate the HomeDef's inherited Contains Association

Constraints in OCL
[12] {home.contents->includesAll (factory)}

Constraints in English
[13] All of the FinderDef metaobjects that populate this Association also populate the HomeDef's inherited Contains Association

Constraints in OCL
[13] {home.contents->includesAll (finder)}

Figure 8-17 Implicit Derived Containments with HomeDef as the Composite

8.1.2.3 *ValueDef Constraints EventsDef*

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 the *Component Model* chapter.

Figure 8-18 expresses the *ValueDef* constraints formally. Note that it uses an OCL technique of defining a side-effect free operation to support recursion, which is required to traverse the transitive closure of a *ValueDefÕs* inheritance hierarchy.

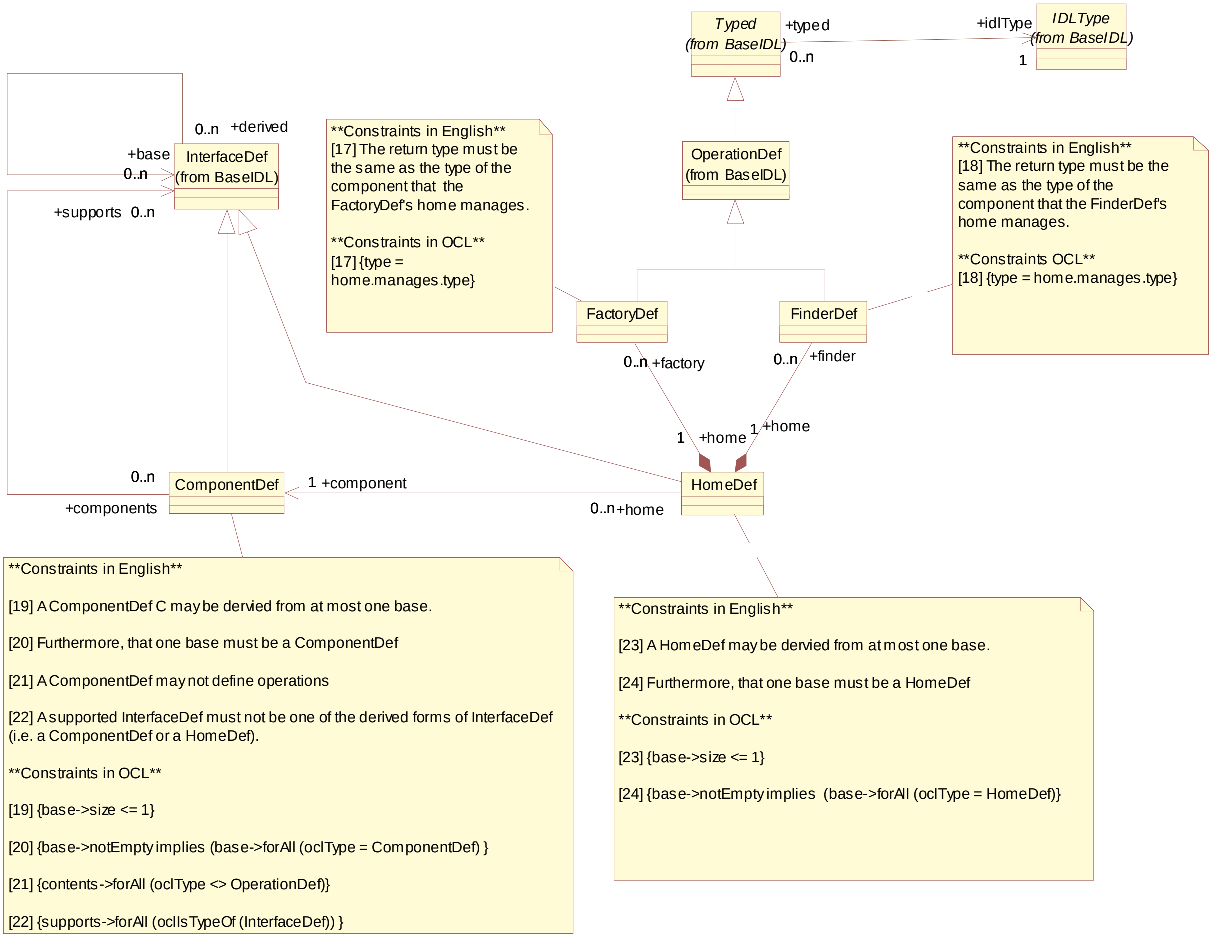
The Component IDL metamodel is changed to introduce the new metatype **eventtype**. This metatype is introduced as a metaclass *EventDef*, which is a specialization of *ValueDef*. Inheritance for instances of *EventDef* is allowed from instances of *ValueDef* and *EventDef*; however, instances of *ValueDef* are not allowed to inherit from instances of *EventDef*.

Figure 8-18 The new metatype **eventtype**

The former metaclass *EventDef* as contained in the metamodel for *ComponentIDL* in orbos/99-07-02 is renamed to *EventPortDef*. The Association between *EventPortDef* and *ValueDef* is removed. Instead, there is a similar Association defined between *EventPortDef* and *EventDef*. The metamodel for Component IDL is shown in Figure 8-12 on page 8-15.

8.1.2.4 Addtional Type and Inheritance Constraints

Figure 8-19 and Figure 8-20 define additional constraints on the *ComponentDef*, *HomeDef*, *FactoryDef*, and *FinderDef* metaclasses.



Constraints in English
[17] The return type must be the same as the type of the component that the FactoryDef's home manages.

Constraints in OCL
[17] {type = home.manages.type}

Constraints in English
[18] The return type must be the same as the type of the component that the FinderDef's home manages.

Constraints OCL
[18] {type = home.manages.type}

Constraints in English

[19] A ComponentDef C may be dervied from at most one base.

[20] Furthermore, that one base must be a ComponentDef

[21] A ComponentDef may not define operations

[22] A supported InterfaceDef must not be one of the derived forms of InterfaceDef (i.e. a ComponentDef or a HomeDef).

Constraints in OCL

[19] {base->size <= 1}

[20] {base->notEmpty implies (base->forAll (oclType = ComponentDef) }

[21] {contents->forAll (oclType <> OperationDef)}

[22] {supports->forAll (oclIsTypeOf (InterfaceDef)) }

Constraints in English

[23] A HomeDef may be dervied from at most one base.

[24] Furthermore, that one base must be a HomeDef

Constraints in OCL

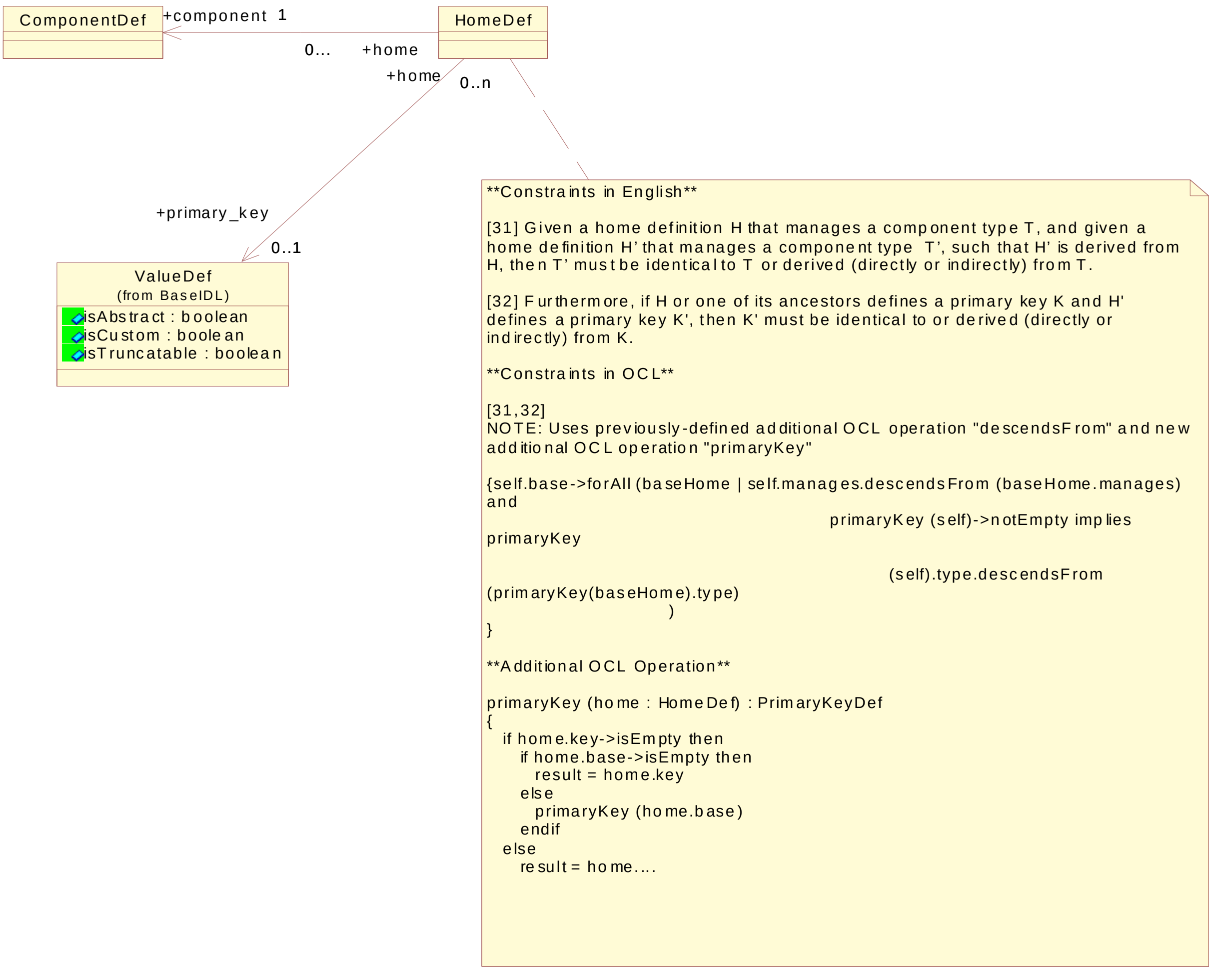
[23] {base->size <= 1}

[24] {base->notEmpty implies (base->forAll (oclType = HomeDef)}

Figure 8-19 Addtional Component, Home, Factory, and Finder Constraints

Note Ð Removed by the Components December 2000 FTF (see ptc/01-11-03, page 70-601).

Figure 8-20 ~~Primary Key Constraints~~



Constraints in English

[31] Given a home definition H that manages a component type T, and given a home definition H' that manages a component type T', such that H' is derived from H, then T' must be identical to T or derived (directly or indirectly) from T.

[32] Furthermore, if H or one of its ancestors defines a primary key K and H' defines a primary key K', then K' must be identical to or derived (directly or indirectly) from K.

Constraints in OCL

[31,32]
NOTE: Uses previously-defined additional OCL operation "descendsFrom" and new additional OCL operation "primaryKey"

```
{self.base->forAll (baseHome | self.manages.descendsFrom (baseHome.manages)
and
                                                  primaryKey (self)->notEmpty implies
primaryKey
                                                           (self).type.descendsFrom
(primaryKey(baseHome).type)
                         )
}
```

Additional OCL Operation

```
primaryKey (home : HomeDef) : PrimaryKeyDef
{
  if home.key->isEmpty then
     if home.base->isEmpty then
       result = home.key
     else
       primaryKey (home.base)
     endif
  else
     result = home....
```

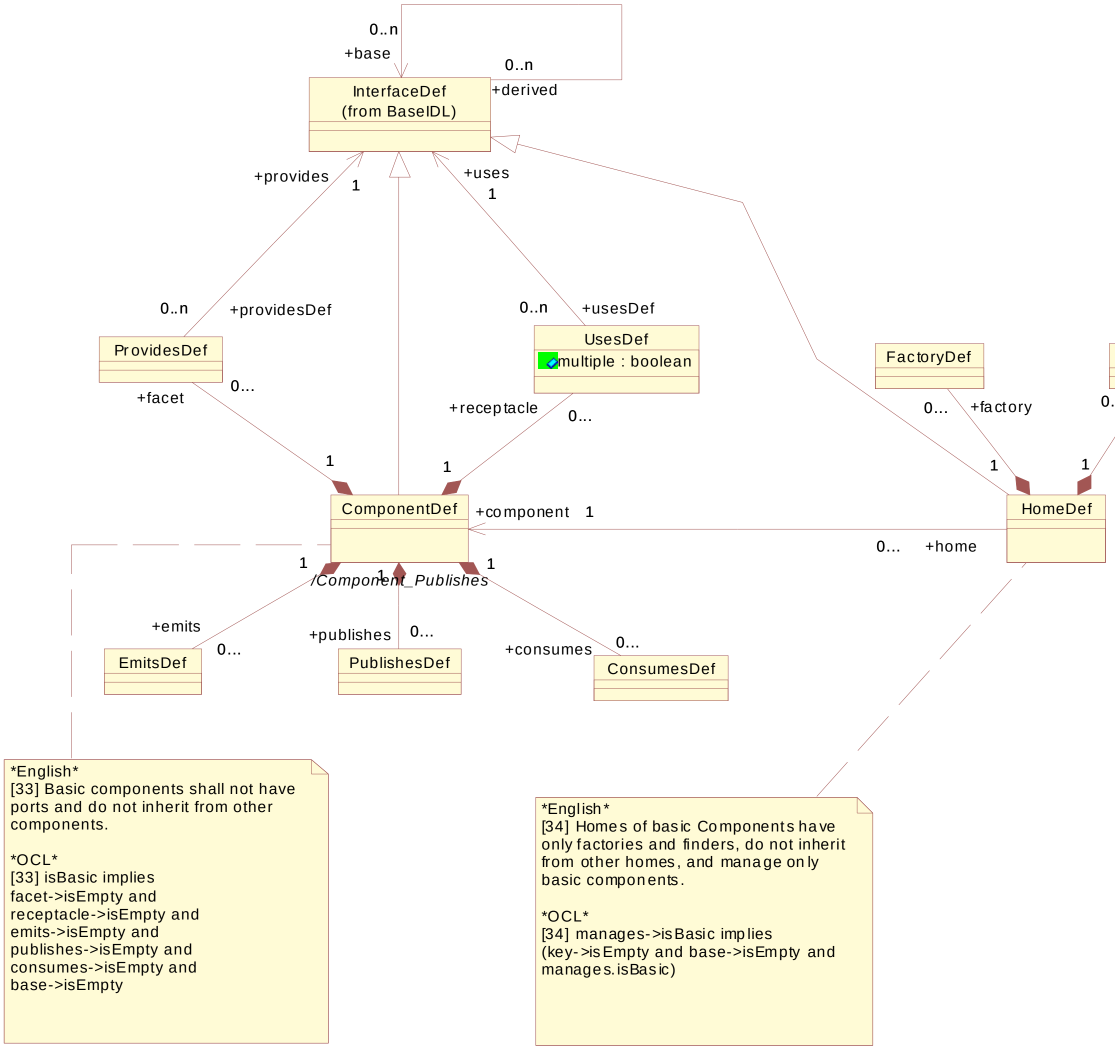
Figure 8-21 Home and Primary Key Constraints

8.1.2.5 Constraints on Basic Components

The CORBA Component Model defines the notion of *basic* components. The *ComponentDef* metaclass has an attribute named *isBasic*. The fact that a component is basic can actually be computed from the component declaration--if the component observes certain constraints, it is basic. Thus, strictly speaking, the *isBasic* attribute is not necessary. However, the attribute greatly simplifies the process of determining whether a component definition is basic.

Given the circumstances, it would seem appropriate to define the *isBasic* attribute as a derived one. In the MOF and UML, *isDerived* is an attribute of *Attribute* that indicates that the information can be computed from other information in the model. However, the XMI standard specifies that the state of derived attributes is not deposited in XMI/XML streams representing models. Thus, if the *isBasic* attribute were marked as derived, the state of the attribute would not appear in XMI streams representing CORBA-based object models. The submitters have therefore decided not to mark the *isBasic* attribute of *ComponentDef* as derived.

The constraints on basic components are modeled formally as shown in Figure 8-22.



English
[33] Basic components shall not have ports and do not inherit from other components.

OCL
[33] isBasic implies
facet->isEmpty and
receptacle->isEmpty and
emits->isEmpty and
publishes->isEmpty and
consumes->isEmpty and
base->isEmpty

English
[34] Homes of basic Components have only factories and finders, do not inherit from other homes, and manage only basic components.

OCL
[34] manages->isBasic implies
(key->isEmpty and base->isEmpty and
manages.isBasic)

Figure 8-22 Constraints on Basic Components

8.2 Conformance Criteria

This chapter identifies the conformance points required for compliant implementations of the interface repository metamodel architecture.

8.2.1 Conformance Points

In the previous section, the MOF metamodel of the Interface Repository is defined. The following section defines the XMI format for the exchange of Interface Repository metadata and the IDL for a MOF-compliant Interface Repository. Support for the generation and consumption of the XMI metadata and for the MOF-compliant IDL is optional.

8.3 MOF DTDs and IDL for the Interface Repository Metamodel

The XMI DTDs and IDL for the Interface Repository metamodel are presented in this section. The DTDs are generated by applying the MOF-XML mapping defined by the XMI specification to the MOF-compliant metamodel described in Section 8.1, “Introduction,” on page 8-1. The IDL is generated by applying the MOF-IDL mapping defined in the MOF specification to the metamodels and was validated using the IDL compilers.

The IDL requires the inclusion of the reflective interfaces defined in ad/99-07-03 as part of the MOF 1.3 specification.

8.3.1 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 "1.1"
            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*, XMI.import*)>

<!--
_______________________________________________________________ --
>
<!--                                                              -->
<!-- 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
               'href CDATA #IMPLIED xmi.idref IDREF #IMPLIED
xml:link
                CDATA #IMPLIED xlink:inline (true|false) #IMPLIED
                xlink:actuate (show|user) #IMPLIED xlink:content-
role
               CDATA #IMPLIED xlink:title CDATA #IMPLIED xlink:show
                (embed|replace|new) #IMPLIED xlink:behavior 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.import identifies imported metamodel(s)
-->
<!--
_______________________________________________________________ --
>

<!ELEMENT XMI.import ANY>
<!ATTLIST XMI.import %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 #IMPLIED>

<!--
_______________________________________________________________ --
>
<!--                                                                -->
<!-- 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.CorbaTcUlongLong|
            XMI.CorbaTcObjRef|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.CorbaTcUlongLong EMPTY>
<!ELEMENT XMI.CorbaTcLongDouble EMPTY>


<!ATTLIST XMI xmlns:BaseIDL CDATA #IMPLIED>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                              -->
<!-- METAMODEL PACKAGE: BaseIDL                                    -
->
<!--                                                              -->
<!--
_______________________________________________________________ --
>

<!ENTITY % BaseIDL: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_LONGDOUBLE|
PK_WSTRING|PK_TYPECODE|PK_WCHAR|PK_PRINCIPAL|PK_STRING|
   PK_ULONGLONG|PK_OBJREF| PK_LONGLONG)'>

<!ENTITY % BaseIDL:ParameterMode '(PARAM_IN|
PARAM_OUT|PARAM_INOUT)'>

<!ENTITY % BaseIDL:DefinitionKind '(DK_NONE|DK_ALL|DK_ATTRIBUTE|

DK_CONSTANT|DK_EXCEPTION|DK_INTERFACE|DK_MODULE|DK_OPERATION|DK_TY
PEDEF|
   DK_ALIAS|DK_STRUCT|DK_UNION|
DK_FIXED|DK_ENUM|DK_PRIMITIVE|DK_STRING|
   DK_SEQUENCE|DK_WSTRING|DK_ARRAY|DK_REPOSITORY)'>

<!--
_______________________________________________________________ --
>
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Typed                                -
->
<!--                                                              -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:Typed.idlType (BaseIDL:IDLType)*>

<!ENTITY % BaseIDL:TypedFeatures 'XMI.extension |
   BaseIDL:Typed.idlType'>

<!ENTITY % BaseIDL:TypedAtts '%XMI.element.att; %XMI.link.att;
   idlType IDREFS #IMPLIED'>

<!ELEMENT BaseIDL:Typed (%BaseIDL:TypedFeatures;)*>

<!ATTLIST BaseIDL:Typed %BaseIDL:TypedAtts;>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.ParameterDef
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ParameterDef.identifier
(#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL:ParameterDef.direction EMPTY>

<!ATTLIST BaseIDL:ParameterDef.direction xmi.value
%BaseIDL:ParameterMode; #REQUIRED>

<!ENTITY % BaseIDL:ParameterDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:ParameterDef.identifier |
   BaseIDL:ParameterDef.direction'>

<!ENTITY % BaseIDL:ParameterDefAtts '%BaseIDL:TypedAtts;
   identifier CDATA #IMPLIED
   direction %BaseIDL:ParameterMode; #IMPLIED'>

<!ELEMENT BaseIDL:ParameterDef (%BaseIDL:ParameterDefFeatures;)*>

<!ATTLIST BaseIDL:ParameterDef %BaseIDL:ParameterDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.Contained                           -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:Contained.definedIn (BaseIDL:Container)*>

<!ELEMENT BaseIDL:Contained.identifier (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL:Contained.repositoryId (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL:Contained.version (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:ContainedFeatures 'XMI.extension |
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version'>
```

```
<!ENTITY % BaseIDL:ContainedAtts '%XMI.element.att; %XMI.link.att;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED'>

<!ELEMENT BaseIDL:Contained (%BaseIDL:ContainedFeatures;)*>

<!ATTLIST BaseIDL:Contained %BaseIDL:ContainedAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.ConstantDef
-->
<!--                                                              -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ConstantDef.constValue (XMI.any)>

<!ENTITY % BaseIDL:ConstantDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version |
   BaseIDL:ConstantDef.constValue'>

<!ENTITY % BaseIDL:ConstantDefAtts '%BaseIDL:TypedAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED'>

<!ELEMENT BaseIDL:ConstantDef (%BaseIDL:ConstantDefFeatures;)*>

<!ATTLIST BaseIDL:ConstantDef %BaseIDL:ConstantDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                              -->
<!-- METAMODEL CLASS: BaseIDL.Container                            -
->
<!--                                                              -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:Container.contents (BaseIDL:Contained|
   BaseIDL:ConstantDef|BaseIDL:TypedefDef|BaseIDL:StructDef|
   BaseIDL:UnionDef|BaseIDL:EnumDef|BaseIDL:AliasDef|
```

```
   BaseIDL:ValueBoxDef|BaseIDL:Container|BaseIDL:ModuleDef|

BaseIDL:InterfaceDef|ComponentIDL:ComponentDef|ComponentIDL:HomeDe
f|
   BaseIDL:ValueDef|ComponentIDL:EventDef|CIF:ComponentImplDef|
   CIF:HomeImplDef|BaseIDL:ValueMemberDef|

BaseIDL:OperationDef|ComponentIDL:FactoryDef|ComponentIDL:FinderDe
f|
   BaseIDL:ExceptionDef|BaseIDL:AttributeDef|
   CIF:ArtifactDef|CIF:SegmentDef|ComponentIDL:ProvidesDef|

ComponentIDL:UsesDef|ComponentIDL:EventPortDef|ComponentIDL:EmitsD
ef|
   ComponentIDL:ConsumesDef|ComponentIDL:PublishesDef)*>

<!ENTITY % BaseIDL:ContainerFeatures '%BaseIDL:ContainedFeatures; |
   BaseIDL:Container.contents'>

<!ENTITY % BaseIDL:ContainerAtts '%BaseIDL:ContainedAtts;'>

<!ELEMENT BaseIDL:Container (%BaseIDL:ContainerFeatures;)*>

<!ATTLIST BaseIDL:Container %BaseIDL:ContainerAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.ModuleDef                          -
->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ModuleDef.prefix (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:ModuleDefFeatures '%BaseIDL:ContainerFeatures; |
   BaseIDL:ModuleDef.prefix'>

<!ENTITY % BaseIDL:ModuleDefAtts '%BaseIDL:ContainerAtts;
   prefix CDATA #IMPLIED'>

<!ELEMENT BaseIDL:ModuleDef (%BaseIDL:ModuleDefFeatures;)*>

<!ATTLIST BaseIDL:ModuleDef %BaseIDL:ModuleDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.IDLType                            -
->
<!--                                                            -->
```

```
<!--
_______________________________________________________________ --
>

<!ENTITY % BaseIDL:IDLTypeFeatures 'XMI.extension'>

<!ENTITY % BaseIDL:IDLTypeAtts '%XMI.element.att; %XMI.link.att;'>

<!ELEMENT BaseIDL:IDLType (%BaseIDL:IDLTypeFeatures;)*>

<!ATTLIST BaseIDL:IDLType %BaseIDL:IDLTypeAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.TypedefDef
-->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ENTITY % BaseIDL:TypedefDefFeatures '%BaseIDL:IDLTypeFeatures; |
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version'>

<!ENTITY % BaseIDL:TypedefDefAtts '%BaseIDL:IDLTypeAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED'>

<!ELEMENT BaseIDL:TypedefDef (%BaseIDL:TypedefDefFeatures;)*>

<!ATTLIST BaseIDL:TypedefDef %BaseIDL:TypedefDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.InterfaceDef
-->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:InterfaceDef.base (BaseIDL:InterfaceDef)*>

<!ELEMENT BaseIDL:InterfaceDef.isAbstract EMPTY>
```

```
<!ATTLIST BaseIDL:InterfaceDef.isAbstract xmi.value (true|false)
#REQUIRED>

<!ELEMENT BaseIDL:InterfaceDef.isLocal EMPTY>

<!ATTLIST BaseIDL:InterfaceDef.isLocal xmi.value (true|false)
#REQUIRED>

<!ENTITY % BaseIDL:InterfaceDefFeatures '%BaseIDL:IDLTypeFeatures;
|
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version |
   BaseIDL:Container.contents |
   BaseIDL:InterfaceDef.base |
   BaseIDL:InterfaceDef.isAbstract |
   BaseIDL:InterfaceDef.isLocal'>

<!ENTITY % BaseIDL:InterfaceDefAtts '%BaseIDL:IDLTypeAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED
   base IDREFS #IMPLIED
   isAbstract (true|false) #IMPLIED
   isLocal (true|false) #IMPLIED'>

<!ELEMENT BaseIDL:InterfaceDef (%BaseIDL:InterfaceDefFeatures;)*>

<!ATTLIST BaseIDL:InterfaceDef %BaseIDL:InterfaceDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.Field                               -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:Field.identifier (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:FieldFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:Field.identifier'>

<!ENTITY % BaseIDL:FieldAtts '%BaseIDL:TypedAtts;
   identifier CDATA #IMPLIED'>

<!ELEMENT BaseIDL:Field (%BaseIDL:FieldFeatures;)*>

<!ATTLIST BaseIDL:Field %BaseIDL:FieldAtts;>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.StructDef                          -
->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:StructDef.members (BaseIDL:Field)*>

<!ENTITY % BaseIDL:StructDefFeatures '%BaseIDL:TypedefDefFeatures;
|
   BaseIDL:StructDef.members'>

<!ENTITY % BaseIDL:StructDefAtts '%BaseIDL:TypedefDefAtts;'>

<!ELEMENT BaseIDL:StructDef (%BaseIDL:StructDefFeatures;)*>

<!ATTLIST BaseIDL:StructDef %BaseIDL:StructDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.UnionDef                           -
->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:UnionDef.discriminatorType (BaseIDL:IDLType)*>

<!ELEMENT BaseIDL:UnionDef.unionMembers (BaseIDL:UnionField)*>

<!ENTITY % BaseIDL:UnionDefFeatures '%BaseIDL:TypedefDefFeatures; |
   BaseIDL:UnionDef.discriminatorType |
   BaseIDL:UnionDef.unionMembers'>

<!ENTITY % BaseIDL:UnionDefAtts '%BaseIDL:TypedefDefAtts;
   discriminatorType IDREFS #IMPLIED'>

<!ELEMENT BaseIDL:UnionDef (%BaseIDL:UnionDefFeatures;)*>

<!ATTLIST BaseIDL:UnionDef %BaseIDL:UnionDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.EnumDef                            -
->
```

```
<!--                                                             -->
<!--
________________________________________________________________ --
>

<!ELEMENT BaseIDL:EnumDef.members (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:EnumDefFeatures '%BaseIDL:TypedefDefFeatures; |
   BaseIDL:EnumDef.members'>

<!ENTITY % BaseIDL:EnumDefAtts '%BaseIDL:TypedefDefAtts;
   members CDATA #IMPLIED'>

<!ELEMENT BaseIDL:EnumDef (%BaseIDL:EnumDefFeatures;)*>

<!ATTLIST BaseIDL:EnumDef %BaseIDL:EnumDefAtts;>

<!--
________________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.AliasDef                             -
->
<!--                                                             -->
<!--
________________________________________________________________ --
>

<!ENTITY % BaseIDL:AliasDefFeatures '%BaseIDL:TypedefDefFeatures; |
   BaseIDL:Typed.idlType'>

<!ENTITY % BaseIDL:AliasDefAtts '%BaseIDL:TypedefDefAtts;
   idlType IDREFS #IMPLIED'>

<!ELEMENT BaseIDL:AliasDef (%BaseIDL:AliasDefFeatures;)*>

<!ATTLIST BaseIDL:AliasDef %BaseIDL:AliasDefAtts;>

<!--
________________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.StringDef                            -
->
<!--                                                             -->
<!--
________________________________________________________________ --
>

<!ELEMENT BaseIDL:StringDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:StringDefFeatures '%BaseIDL:IDLTypeFeatures; |
   BaseIDL:StringDef.bound'>

<!ENTITY % BaseIDL:StringDefAtts '%BaseIDL:IDLTypeAtts;
```

```
   bound CDATA #IMPLIED'>

<!ELEMENT BaseIDL:StringDef (%BaseIDL:StringDefFeatures;)*>

<!ATTLIST BaseIDL:StringDef %BaseIDL:StringDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.WStringDef
-->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:WStringDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:WStringDefFeatures '%BaseIDL:IDLTypeFeatures; |
   BaseIDL:WStringDef.bound'>

<!ENTITY % BaseIDL:WStringDefAtts '%BaseIDL:IDLTypeAtts;
   bound CDATA #IMPLIED'>

<!ELEMENT BaseIDL:WStringDef (%BaseIDL:WStringDefFeatures;)*>

<!ATTLIST BaseIDL:WStringDef %BaseIDL:WStringDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.FixedDef                            -
->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:FixedDef.digits (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL:FixedDef.scale (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:FixedDefFeatures '%BaseIDL:IDLTypeFeatures; |
   BaseIDL:FixedDef.digits |
   BaseIDL:FixedDef.scale'>

<!ENTITY % BaseIDL:FixedDefAtts '%BaseIDL:IDLTypeAtts;
   digits CDATA #IMPLIED
   scale CDATA #IMPLIED'>

<!ELEMENT BaseIDL:FixedDef (%BaseIDL:FixedDefFeatures;)*>

<!ATTLIST BaseIDL:FixedDef %BaseIDL:FixedDefAtts;>
```

```
<!--
______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.SequenceDef
-->
<!--                                                            -->
<!--
______________________________________________________________ --
>

<!ELEMENT BaseIDL:SequenceDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:SequenceDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:SequenceDef.bound'>

<!ENTITY % BaseIDL:SequenceDefAtts '%BaseIDL:TypedAtts;
   bound CDATA #IMPLIED'>

<!ELEMENT BaseIDL:SequenceDef (%BaseIDL:SequenceDefFeatures;)*>

<!ATTLIST BaseIDL:SequenceDef %BaseIDL:SequenceDefAtts;>

<!--
______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.ArrayDef                           -
->
<!--                                                            -->
<!--
______________________________________________________________ --
>

<!ELEMENT BaseIDL:ArrayDef.bound (#PCDATA|XMI.reference)*>

<!ENTITY % BaseIDL:ArrayDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:ArrayDef.bound'>

<!ENTITY % BaseIDL:ArrayDefAtts '%BaseIDL:TypedAtts;
   bound CDATA #IMPLIED'>

<!ELEMENT BaseIDL:ArrayDef (%BaseIDL:ArrayDefFeatures;)*>

<!ATTLIST BaseIDL:ArrayDef %BaseIDL:ArrayDefAtts;>

<!--
______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.PrimitiveDef
-->
<!--                                                            -->
```

```
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:PrimitiveDef.kind EMPTY>

<!ATTLIST BaseIDL:PrimitiveDef.kind xmi.value
%BaseIDL:PrimitiveKind; #REQUIRED>

<!ENTITY % BaseIDL:PrimitiveDefFeatures '%BaseIDL:IDLTypeFeatures;
|
   BaseIDL:PrimitiveDef.kind'>

<!ENTITY % BaseIDL:PrimitiveDefAtts '%BaseIDL:IDLTypeAtts;
   kind %BaseIDL:PrimitiveKind; #IMPLIED'>

<!ELEMENT BaseIDL:PrimitiveDef (%BaseIDL:PrimitiveDefFeatures;)*>

<!ATTLIST BaseIDL:PrimitiveDef %BaseIDL:PrimitiveDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.UnionField
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:UnionField.identifier (#PCDATA|XMI.reference)*>

<!ELEMENT BaseIDL:UnionField.label (XMI.any)>

<!ENTITY % BaseIDL:UnionFieldFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:UnionField.identifier |
   BaseIDL:UnionField.label'>

<!ENTITY % BaseIDL:UnionFieldAtts '%BaseIDL:TypedAtts;
   identifier CDATA #IMPLIED'>

<!ELEMENT BaseIDL:UnionField (%BaseIDL:UnionFieldFeatures;)*>

<!ATTLIST BaseIDL:UnionField %BaseIDL:UnionFieldAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.ValueMemberDef
-->
<!--                                                             -->
```

```
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ValueMemberDef.isPublicMember EMPTY>

<!ATTLIST BaseIDL:ValueMemberDef.isPublicMember xmi.value
(true|false) #REQUIRED>

<!ENTITY % BaseIDL:ValueMemberDefFeatures '%BaseIDL:TypedFeatures;
|
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version |
   BaseIDL:ValueMemberDef.isPublicMember'>

<!ENTITY % BaseIDL:ValueMemberDefAtts '%BaseIDL:TypedAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED
   isPublicMember (true|false) #IMPLIED'>

<!ELEMENT BaseIDL:ValueMemberDef
(%BaseIDL:ValueMemberDefFeatures;)*>

<!ATTLIST BaseIDL:ValueMemberDef %BaseIDL:ValueMemberDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.ValueDef                            -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ValueDef.interfaceDef (BaseIDL:InterfaceDef)*>

<!ELEMENT BaseIDL:ValueDef.base (BaseIDL:ValueDef)*>

<!ELEMENT BaseIDL:ValueDef.abstractBase (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>

<!ENTITY % BaseIDL:ValueDefFeatures '%BaseIDL:ContainerFeatures; |
   BaseIDL:ValueDef.interfaceDef |
   BaseIDL:ValueDef.base |
   BaseIDL:ValueDef.abstractBase |
   BaseIDL:ValueDef.isAbstract |
   BaseIDL:ValueDef.isCustom |
   BaseIDL:ValueDef.isTruncatable'>

<!ENTITY % BaseIDL:ValueDefAtts '%BaseIDL:ContainerAtts;
   interfaceDef IDREFS #IMPLIED
   base IDREFS #IMPLIED
   abstractBase IDREFS #IMPLIED
   isAbstract (true|false) #IMPLIED
   isCustom (true|false) #IMPLIED
   isTruncatable (true|false) #IMPLIED'>

<!ELEMENT BaseIDL:ValueDef (%BaseIDL:ValueDefFeatures;)*>

<!ATTLIST BaseIDL:ValueDef %BaseIDL:ValueDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.ValueBoxDef
-->
<!--                                                            -->
<!--
_______________________________________________________________ --
>

<!ENTITY % BaseIDL:ValueBoxDefFeatures
'%BaseIDL:TypedefDefFeatures;'>

<!ENTITY % BaseIDL:ValueBoxDefAtts '%BaseIDL:TypedefDefAtts;'>

<!ELEMENT BaseIDL:ValueBoxDef (%BaseIDL:ValueBoxDefFeatures;)*>

<!ATTLIST BaseIDL:ValueBoxDef %BaseIDL:ValueBoxDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                            -->
<!-- METAMODEL CLASS: BaseIDL.OperationDef
-->
<!--                                                            -->
```

```
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:OperationDef.exceptionDef
(BaseIDL:ExceptionDef)*>

<!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)*>

<!ENTITY % BaseIDL:OperationDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version |
   BaseIDL:OperationDef.exceptionDef |
   BaseIDL:OperationDef.isOneway |
   BaseIDL:OperationDef.parameters |
   BaseIDL:OperationDef.contexts'>

<!ENTITY % BaseIDL:OperationDefAtts '%BaseIDL:TypedAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED
   exceptionDef IDREFS #IMPLIED
   isOneway (true|false) #IMPLIED
   contexts CDATA #IMPLIED'>

<!ELEMENT BaseIDL:OperationDef (%BaseIDL:OperationDefFeatures;)*>

<!ATTLIST BaseIDL:OperationDef %BaseIDL:OperationDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: BaseIDL.ExceptionDef
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:ExceptionDef.typeCode (XMI.CorbaTypeCode)>

<!ELEMENT BaseIDL:ExceptionDef.members (BaseIDL:Field)*>
```

```
<!ENTITY % BaseIDL:ExceptionDefFeatures
'%BaseIDL:ContainedFeatures; |
   BaseIDL:ExceptionDef.typeCode |
   BaseIDL:ExceptionDef.members'>

<!ENTITY % BaseIDL:ExceptionDefAtts '%BaseIDL:ContainedAtts;'>

<!ELEMENT BaseIDL:ExceptionDef (%BaseIDL:ExceptionDefFeatures;)*>

<!ATTLIST BaseIDL:ExceptionDef %BaseIDL:ExceptionDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                                  -->
<!-- METAMODEL CLASS: BaseIDL.AttributeDef
-->
<!--                                                                  -->
<!--
_______________________________________________________________ --
>

<!ELEMENT BaseIDL:AttributeDef.setException
(BaseIDL:ExceptionDef)*>

<!ELEMENT BaseIDL:AttributeDef.getException
(BaseIDL:ExceptionDef)*>

<!ELEMENT BaseIDL:AttributeDef.isReadonly EMPTY>

<!ATTLIST BaseIDL:AttributeDef.isReadonly xmi.value (true|false)
#REQUIRED>

<!ENTITY % BaseIDL:AttributeDefFeatures '%BaseIDL:TypedFeatures; |
   BaseIDL:Contained.definedIn |
   BaseIDL:Contained.identifier |
   BaseIDL:Contained.repositoryId |
   BaseIDL:Contained.version |
   BaseIDL:AttributeDef.setException |
   BaseIDL:AttributeDef.getException |
   BaseIDL:AttributeDef.isReadonly'>

<!ENTITY % BaseIDL:AttributeDefAtts '%BaseIDL:TypedAtts;
   definedIn IDREFS #IMPLIED
   identifier CDATA #IMPLIED
   repositoryId CDATA #IMPLIED
   version CDATA #IMPLIED
   setException IDREFS #IMPLIED
   getException IDREFS #IMPLIED
   isReadonly (true|false) #IMPLIED'>

<!ELEMENT BaseIDL:AttributeDef (%BaseIDL:AttributeDefFeatures;)*>

<!ATTLIST BaseIDL:AttributeDef %BaseIDL:AttributeDefAtts;>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL PACKAGE: CIF                                       -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ENTITY % CIF:ComponentCategory
'(PROCESS|SESSION|ENTITY|SERVICE)'>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: CIF:ArtifactDef                             -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ENTITY % CIF:ArtifactDefFeatures '%BaseIDL:ContainedFeatures;'>

<!ENTITY % CIF:ArtifactDefAtts '%BaseIDL:ContainedAtts;'>

<!ELEMENT CIF:ArtifactDef (%CIF:ArtifactDefFeatures;)*>

<!ATTLIST CIF:ArtifactDef %CIF:ArtifactDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: CIF:SegmentDef                              -
->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ELEMENT CIF:SegmentDef.artifact (CIF:ArtifactDef)*>

<!ELEMENT CIF:SegmentDef.features
(ComponentIDL:ComponentFeature)*>

<!ELEMENT CIF:SegmentDef.policies (CIF:Policy)*>

<!ELEMENT CIF:SegmentDef.isSerialized EMPTY>

<!ATTLIST CIF:SegmentDef.isSerialized xmi.value (true|false)
#REQUIRED>
```

```
<!ENTITY % CIF:SegmentDefFeatures '%BaseIDL:ContainedFeatures; |
   CIF:SegmentDef.artifact |
   CIF:SegmentDef.features |
   CIF:SegmentDef.policies |
   CIF:SegmentDef.isSerialized'>

<!ENTITY % CIF:SegmentDefAtts '%BaseIDL:ContainedAtts;
   artifact IDREFS #IMPLIED
   features IDREFS #IMPLIED
   policies IDREFS #IMPLIED
   isSerialized (true|false) #IMPLIED'>

<!ELEMENT CIF:SegmentDef (%CIF:SegmentDefFeatures;)*>

<!ATTLIST CIF:SegmentDef %CIF:SegmentDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                                -->
<!-- METAMODEL CLASS: CIF:ComponentImplDef
-->
<!--                                                                -->
<!--
_______________________________________________________________ --
>

<!ELEMENT CIF:ComponentImplDef.component
(ComponentIDL:ComponentDef)*>

<!ELEMENT CIF:ComponentImplDef.category EMPTY>

<!ATTLIST CIF:ComponentImplDef.category xmi.value
%CIF:ComponentCategory; #REQUIRED>

<!ENTITY % CIF:ComponentImplDefFeatures
'%BaseIDL:ContainerFeatures; |
   CIF:ComponentImplDef.component |
   CIF:ComponentImplDef.category'>

<!ENTITY % CIF:ComponentImplDefAtts '%BaseIDL:ContainerAtts;
   component IDREFS #IMPLIED
   category %CIF:ComponentCategory; #IMPLIED'>

<!ELEMENT CIF:ComponentImplDef (%CIF:ComponentImplDefFeatures;)*>

<!ATTLIST CIF:ComponentImplDef %CIF:ComponentImplDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                                -->
<!-- METAMODEL CLASS: CIF:Policy                                     -
->
```

```
<!--                                                            -->
<!--
_______________________________________________________________  --
>

<!ENTITY % CIF:PolicyFeatures 'XMI.extension'>

<!ENTITY % CIF:PolicyAtts '%XMI.element.att; %XMI.link.att;'>

<!ELEMENT CIF:Policy (%CIF:PolicyFeatures;)*>

<!ATTLIST CIF:Policy %CIF:PolicyAtts;>

<!--
_______________________________________________________________  --
>
<!--                                                            -->
<!-- METAMODEL CLASS: CIF:HomeImplDef                             -
->
<!--                                                            -->
<!--
_______________________________________________________________  --
>

<!ELEMENT CIF:HomeImplDef.home (ComponentIDL:HomeDef)*>

<!ELEMENT CIF:HomeImplDef.component_impl (CIF:ComponentImplDef)*>

<!ENTITY % CIF:HomeImplDefFeatures '%BaseIDL:ContainerFeatures; |
   CIF:HomeImplDef.home |
   CIF:HomeImplDef.component_impl'>

<!ENTITY % CIF:HomeImplDefAtts '%BaseIDL:ContainerAtts;
   home IDREFS #IMPLIED
   component_impl IDREFS #IMPLIED'>

<!ELEMENT CIF:HomeImplDef (%CIF:HomeImplDefFeatures;)*>

<!ATTLIST CIF:HomeImplDef %CIF:HomeImplDefAtts;>

<!--
_______________________________________________________________  --
>
<!--                                                            -->
<!-- METAMODEL PACKAGE: ComponentIDL                              -
->
<!--                                                            -->
<!--
_______________________________________________________________  --
>

<!--
_______________________________________________________________  --
>
<!--                                                            -->
```

```
<!-- METAMODEL CLASS: ComponentIDL:ComponentFeature
-->
<!--                                                          -->
<!--
______________________________________________________________ --
>

<!ENTITY % ComponentIDL:ComponentFeatureFeatures 'XMI.extension'>

<!ENTITY % ComponentIDL:ComponentFeatureAtts '%XMI.element.att;
%XMI.link.att;'>

<!ELEMENT ComponentIDL:ComponentFeature
(%ComponentIDL:ComponentFeatureFeatures;)*>

<!ATTLIST ComponentIDL:ComponentFeature
%ComponentIDL:ComponentFeatureAtts;>

<!--
______________________________________________________________ --
>
<!--                                                          -->
<!-- METAMODEL CLASS: ComponentIDL:ComponentDef
-->
<!--                                                          -->
<!--
______________________________________________________________ --
>

<!ELEMENT ComponentIDL:ComponentDef.supports
(BaseIDL:InterfaceDef)*>

<!ENTITY % ComponentIDL:ComponentDefFeatures
'%BaseIDL:InterfaceDefFeatures; |
   ComponentIDL:ComponentDef.supports'>

<!ENTITY % ComponentIDL:ComponentDefAtts
'%BaseIDL:InterfaceDefAtts;
   supports IDREFS #IMPLIED'>

<!ELEMENT ComponentIDL:ComponentDef
(%ComponentIDL:ComponentDefFeatures;)*>

<!ATTLIST ComponentIDL:ComponentDef
%ComponentIDL:ComponentDefAtts;>

<!--
______________________________________________________________ --
>
<!--                                                          -->
<!-- METAMODEL CLASS: ComponentIDL:ProvidesDef
-->
<!--                                                          -->
```

```
<!--
_______________________________________________________________ --
>

<!ELEMENT ComponentIDL:ProvidesDef.provides
(BaseIDL:InterfaceDef)*>

<!ENTITY % ComponentIDL:ProvidesDefFeatures
'%BaseIDL:ContainedFeatures; |
   ComponentIDL:ProvidesDef.provides'>

<!ENTITY % ComponentIDL:ProvidesDefAtts '%BaseIDL:ContainedAtts;
   provides IDREFS #IMPLIED'>

<!ELEMENT ComponentIDL:ProvidesDef
(%ComponentIDL:ProvidesDefFeatures;)*>

<!ATTLIST ComponentIDL:ProvidesDef %ComponentIDL:ProvidesDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                          -->
<!-- METAMODEL CLASS: ComponentIDL:HomeDef
-->
<!--                                                          -->
<!--
_______________________________________________________________ --
>

<!ELEMENT ComponentIDL:HomeDef.component
(ComponentIDL:ComponentDef)*>

<!ELEMENT ComponentIDL:HomeDef.primary_key (BaseIDL:ValueDef)*>

<!ELEMENT ComponentIDL:HomeDef.supports (BaseIDL:InterfaceDef)*>

<!ENTITY % ComponentIDL:HomeDefFeatures
'%BaseIDL:InterfaceDefFeatures; |
   ComponentIDL:HomeDef.component |
   ComponentIDL:HomeDef.primary_key |
   ComponentIDL:HomeDef.supports'>

<!ENTITY % ComponentIDL:HomeDefAtts '%BaseIDL:InterfaceDefAtts;
   component IDREFS #IMPLIED
   primary_key IDREFS #IMPLIED
   supports IDREFS #IMPLIED'>

<!ELEMENT ComponentIDL:HomeDef (%ComponentIDL:HomeDefFeatures;)*>

<!ATTLIST ComponentIDL:HomeDef %ComponentIDL:HomeDefAtts;>

<!--
_______________________________________________________________ --
>
```

```
<!--                                                                  -->
<!-- METAMODEL CLASS: ComponentIDL:FactoryDef
-->
<!--                                                                  -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:FactoryDefFeatures
'%BaseIDL:OperationDefFeatures;'>

<!ENTITY % ComponentIDL:FactoryDefAtts
'%BaseIDL:OperationDefAtts;'>

<!ELEMENT ComponentIDL:FactoryDef
(%ComponentIDL:FactoryDefFeatures;)*>

<!ATTLIST ComponentIDL:FactoryDef %ComponentIDL:FactoryDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                                  -->
<!-- METAMODEL CLASS: ComponentIDL:FinderDef
-->
<!--                                                                  -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:FinderDefFeatures
'%BaseIDL:OperationDefFeatures;'>

<!ENTITY % ComponentIDL:FinderDefAtts '%BaseIDL:OperationDefAtts;'>

<!ELEMENT ComponentIDL:FinderDef
(%ComponentIDL:FinderDefFeatures;)*>

<!ATTLIST ComponentIDL:FinderDef %ComponentIDL:FinderDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                                  -->
<!-- METAMODEL CLASS: ComponentIDL:EventPortDef
-->
<!--                                                                  -->
<!--
_______________________________________________________________ --
>

<!ELEMENT ComponentIDL:EventPortDef.type (ComponentIDL:EventDef)*>

<!ENTITY % ComponentIDL:EventPortDefFeatures
'%BaseIDL:ContainedFeatures; |
```

```
   ComponentIDL:EventPortDef.type'>

<!ENTITY % ComponentIDL:EventPortDefAtts '%BaseIDL:ContainedAtts;
   type IDREFS #IMPLIED'>

<!ELEMENT ComponentIDL:EventPortDef
(%ComponentIDL:EventPortDefFeatures;)*>

<!ATTLIST ComponentIDL:EventPortDef
%ComponentIDL:EventPortDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: ComponentIDL:EmitsDef
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:EmitsDefFeatures
'%ComponentIDL:EventPortDefFeatures;'>

<!ENTITY % ComponentIDL:EmitsDefAtts
'%ComponentIDL:EventPortDefAtts;'>

<!ELEMENT ComponentIDL:EmitsDef
(%ComponentIDL:EmitsDefFeatures;)*>

<!ATTLIST ComponentIDL:EmitsDef %ComponentIDL:EmitsDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: ComponentIDL:ConsumesDef
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:ConsumesDefFeatures
'%ComponentIDL:EventPortDefFeatures;'>

<!ENTITY % ComponentIDL:ConsumesDefAtts
'%ComponentIDL:EventPortDefAtts;'>

<!ELEMENT ComponentIDL:ConsumesDef
(%ComponentIDL:ConsumesDefFeatures;)*>

<!ATTLIST ComponentIDL:ConsumesDef %ComponentIDL:ConsumesDefAtts;>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                               -->
<!-- METAMODEL CLASS: ComponentIDL:UsesDef
-->
<!--                                                               -->
<!--
_______________________________________________________________ --
>

<!ELEMENT ComponentIDL:UsesDef.uses (BaseIDL:InterfaceDef)*>

<!ELEMENT ComponentIDL:UsesDef.multiple EMPTY>

<!ATTLIST ComponentIDL:UsesDef.multiple xmi.value (true|false)
#REQUIRED>

<!ENTITY % ComponentIDL:UsesDefFeatures
'%BaseIDL:ContainedFeatures; |
   ComponentIDL:UsesDef.uses |
   ComponentIDL:UsesDef.multiple'>

<!ENTITY % ComponentIDL:UsesDefAtts '%BaseIDL:ContainedAtts;
   uses IDREFS #IMPLIED
   multiple (true|false) #IMPLIED'>

<!ELEMENT ComponentIDL:UsesDef (%ComponentIDL:UsesDefFeatures;)*>

<!ATTLIST ComponentIDL:UsesDef %ComponentIDL:UsesDefAtts;>

<!--
_______________________________________________________________ --
>
<!--                                                               -->
<!-- METAMODEL CLASS: ComponentIDL:PublishesDef
-->
<!--                                                               -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:PublishesDefFeatures
'%ComponentIDL:EventPortDefFeatures;'>

<!ENTITY % ComponentIDL:PublishesDefAtts
'%ComponentIDL:EventPortDefAtts;'>

<!ELEMENT ComponentIDL:PublishesDef
(%ComponentIDL:PublishesDefFeatures;)*>

<!ATTLIST ComponentIDL:PublishesDef
%ComponentIDL:PublishesDefAtts;>
```

```
<!--
_______________________________________________________________ --
>
<!--                                                             -->
<!-- METAMODEL CLASS: ComponentIDL:EventDef
-->
<!--                                                             -->
<!--
_______________________________________________________________ --
>

<!ENTITY % ComponentIDL:EventDefFeatures
'%BaseIDL:ValueDefFeatures;'>

<!ENTITY % ComponentIDL:EventDefAtts '%BaseIDL:ValueDefAtts;'>

<!ELEMENT ComponentIDL:EventDef
(%ComponentIDL:EventDefFeatures;)*>

<!ATTLIST ComponentIDL:EventDef %ComponentIDL:EventDefAtts;>
```

8.3.2 IDL for the BaseIDL Package

```
#pragma prefix "ccm.omg.org"
#include "Reflective.idl"

module BaseIDL
{
  interface TypedClass;
  interface Typed;
  typedef sequence<Typed> TypedSet;
  interface ParameterDefClass;
  interface ParameterDef;
  typedef sequence<ParameterDef> ParameterDefSet;
  interface ContainedClass;
  interface Contained;
  typedef sequence<Contained> ContainedSet;
  interface ConstantDefClass;
  interface ConstantDef;
  typedef sequence<ConstantDef> ConstantDefSet;
  interface ContainerClass;
  interface Container;
  typedef sequence<Container> ContainerSet;
  interface ModuleDefClass;
  interface ModuleDef;
  typedef sequence<ModuleDef> ModuleDefSet;
  interface IDLTypeClass;
  interface IDLType;
  typedef sequence<IDLType> IDLTypeSet;
  interface TypedefDefClass;
  interface TypedefDef;
  typedef sequence<TypedefDef> TypedefDefSet;
  interface InterfaceDefClass;
  interface InterfaceDef;
  typedef sequence<InterfaceDef> InterfaceDefSet;
  interface FieldClass;
  interface Field;
  typedef sequence<Field> FieldSet;
  interface StructDefClass;
  interface StructDef;
  typedef sequence<StructDef> StructDefSet;
  interface UnionDefClass;
  interface UnionDef;
  typedef sequence<UnionDef> UnionDefSet;
  interface EnumDefClass;
  interface EnumDef;
  typedef sequence<EnumDef> EnumDefSet;
  interface AliasDefClass;
  interface AliasDef;
  typedef sequence<AliasDef> AliasDefSet;
  interface StringDefClass;
```

```
  interface StringDef;
  typedef sequence<StringDef> StringDefSet;
  interface WstringDefClass;
  interface WstringDef;
  typedef sequence<WstringDef> WstringDefSet;
  interface FixedDefClass;
  interface FixedDef;
  typedef sequence<FixedDef> FixedDefSet;
  interface SequenceDefClass;
  interface SequenceDef;
  typedef sequence<SequenceDef> SequenceDefSet;
  interface ArrayDefClass;
  interface ArrayDef;
  typedef sequence<ArrayDef> ArrayDefSet;
  interface PrimitiveDefClass;
  interface PrimitiveDef;
  typedef sequence<PrimitiveDef> PrimitiveDefSet;
  interface UnionFieldClass;
  interface UnionField;
  typedef sequence<UnionField> UnionFieldSet;
  interface ValueMemberDefClass;
  interface ValueMemberDef;
  typedef sequence<ValueMemberDef> ValueMemberDefSet;
  interface ValueDefClass;
  interface ValueDef;
  typedef sequence<ValueDef> ValueDefSet;
  interface ValueBoxDefClass;
  interface ValueBoxDef;
  typedef sequence<ValueBoxDef> ValueBoxDefSet;
  interface OperationDefClass;
  interface OperationDef;
  typedef sequence<OperationDef> OperationDefSet;
  interface ExceptionDefClass;
  interface ExceptionDef;
  typedef sequence<ExceptionDef> ExceptionDefSet;
  interface AttributeDefClass;
  interface AttributeDef;
  typedef sequence<AttributeDef> AttributeDefSet;
  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_LONGDOUBLE,  PK_WSTRING,
PK_TYPECODE, PK_WCHAR, PK_PRINCIPAL, PK_STRING,
PK_ULONGLONG, PK_OBJREF,  PK_LONGLONG};
  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_FIXED, DK_ENUM, DK_PRIMITIVE, DK_STRING, DK_SEQUENCE,
DK_WSTRING, DK_ARRAY, DK_REPOSITORY};
```

```
interface TypedClass : Reflective::RefObject
{
  readonly attribute TypedSet all_of_type_typed;
};

interface Typed : TypedClass
{
  IDLType idl_type ()
    raises (Reflective::MofError);
  void set_idl_type (in IDLType new_value)
    raises (Reflective::MofError);
}; // end of interface Typed

interface ParameterDefClass : TypedClass
{
  readonly attribute ParameterDefSet all_of_type_parameter_def;
  readonly attribute ParameterDefSet all_of_class_parameter_def;
  ParameterDef create_parameter_def (
    in string identifier,
    in ParameterMode direction)
    raises (Reflective::MofError);
};

interface ParameterDef : ParameterDefClass, Typed
{
  string identifier ()
    raises (Reflective::MofError);
  void set_identifier (in string new_value)
    raises (Reflective::MofError);
  ParameterMode direction ()
    raises (Reflective::MofError);
  void set_direction (in ParameterMode new_value)
    raises (Reflective::MofError);
}; // end of interface ParameterDef

interface ContainedClass : Reflective::RefObject
{
  readonly attribute ContainedSet all_of_type_contained;
};

interface Contained : ContainedClass
{
  Container defined_in ()
    raises (Reflective::NotSet, Reflective::MofError);
  void set_defined_in (in Container new_value)
    raises (Reflective::MofError);
  void unset_defined_in ()
    raises (Reflective::MofError);
  string identifier ()
    raises (Reflective::MofError);
  void set_identifier (in string new_value)
```

```
    raises (Reflective::MofError);
  string repository_id ()
    raises (Reflective::MofError);
  void set_repository_id (in string new_value)
    raises (Reflective::MofError);
  string version ()
    raises (Reflective::MofError);
  void set_version (in string new_value)
    raises (Reflective::MofError);
  string absolute_name ()
    raises (Reflective::MofError);
}; // end of interface Contained

interface ConstantDefClass : TypedClass, ContainedClass
{
  readonly attribute ConstantDefSet all_of_type_constant_def;
  readonly attribute ConstantDefSet all_of_class_constant_def;
  ConstantDef create_constant_def (
    in string identifier,
    in string repository_id,
    in string version,
    in any const_value)
    raises (Reflective::MofError);
};

interface ConstantDef : ConstantDefClass, Typed, Contained
{
  any const_value ()
    raises (Reflective::MofError);
  void set_const_value (in any new_value)
    raises (Reflective::MofError);
}; // end of interface ConstantDef

interface ContainerClass : ContainedClass
{
  readonly attribute ContainerSet all_of_type_container;
};

interface Container : ContainerClass, Contained
{
  ContainedSet contents ()
    raises (Reflective::MofError);
  void set_contents (in ContainedSet new_value)
    raises (Reflective::MofError);
  void unset_contents ()
    raises (Reflective::MofError);
  void add_contents (in Contained new_element)
    raises (Reflective::MofError);
  void modify_contents (
    in Contained old_element,
    in Contained new_element)
```

```
    raises (Reflective::NotFound, Reflective::MofError);
  void remove_contents (in Contained old_element)
    raises (Reflective::NotFound, Reflective::MofError);
  Contained lookup_name(
    in string search_name,
    in long levels_to_search,
    in DefinitionKind limit_to_type,
    in boolean exclude_inherited)
    raises (Reflective::MofError);
  Contained lookup(
    in string search_name)
    raises (Reflective::MofError);
  Contained get_filtered_contents(
    in DefinitionKind limit_to_type,
    in boolean include_inherited)
    raises (Reflective::MofError);
}; // end of interface Container

interface ModuleDefClass : ContainerClass
{
  readonly attribute ModuleDefSet all_of_type_module_def;
  readonly attribute ModuleDefSet all_of_class_module_def;
  ModuleDef create_module_def (
    in string identifier,
    in string repository_id,
    in string version,
    in string prefix)
    raises (Reflective::MofError);
};

interface ModuleDef : ModuleDefClass, Container
{
  string prefix ()
    raises (Reflective::MofError);
  void set_prefix (in string new_value)
    raises (Reflective::MofError);
}; // end of interface ModuleDef

interface IDLTypeClass : Reflective::RefObject
{
  readonly attribute IDLTypeSet all_of_type_idltype;
};

interface IDLType : IDLTypeClass
{
  CORBA::TypeCode type_code ()
    raises (Reflective::MofError);
}; // end of interface IDLType

interface TypedefDefClass : IDLTypeClass, ContainedClass
{
```

```
   readonly attribute TypedefDefSet all_of_type_typedef_def;
};

interface TypedefDef : TypedefDefClass, IDLType, Contained
{
}; // end of interface TypedefDef

interface InterfaceDefClass : IDLTypeClass, ContainerClass
{
   readonly attribute InterfaceDefSet all_of_type_interface_def;
   readonly attribute InterfaceDefSet all_of_class_interface_def;
   InterfaceDef create_interface_def (
      in string identifier,
      in string repository_id,
      in string version,
      in boolean is_abstract,
      in boolean is_local)
      raises (Reflective::MofError);
};

interface InterfaceDef : InterfaceDefClass, IDLType, Container
{
   InterfaceDefSet base ()
      raises (Reflective::MofError);
   void set_base (in InterfaceDefSet new_value)
      raises (Reflective::MofError);
   void unset_base ()
      raises (Reflective::MofError);
   void add_base (in InterfaceDef new_element)
      raises (Reflective::MofError);
   void modify_base (
     in InterfaceDef old_element,
     in InterfaceDef new_element)
     raises (Reflective::NotFound, Reflective::MofError);
   void remove_base (in InterfaceDef old_element)
     raises (Reflective::NotFound, Reflective::MofError);
   boolean is_abstract ()
     raises (Reflective::MofError);
   void set_is_abstract (in boolean new_value)
      raises (Reflective::MofError);
   boolean is_local ()
      raises (Reflective::MofError);
   void set_is_local (in boolean new_value)
      raises (Reflective::MofError);
}; // end of interface InterfaceDef

interface FieldClass : TypedClass
{
   readonly attribute FieldSet all_of_type_field;
   readonly attribute FieldSet all_of_class_field;
   Field create_field (
```

```
        in string identifier)
        raises (Reflective::MofError);
};

interface Field : FieldClass, Typed
{
    string identifier ()
        raises (Reflective::MofError);
    void set_identifier (in string new_value)
        raises (Reflective::MofError);
}; // end of interface Field

interface StructDefClass : TypedefDefClass
{
    readonly attribute StructDefSet all_of_type_struct_def;
    readonly attribute StructDefSet all_of_class_struct_def;
    StructDef create_struct_def (
        in string identifier,
        in string repository_id,
        in string version,
        in Field members)
        raises (Reflective::MofError);
};

interface StructDef : StructDefClass, TypedefDef
{
    Field members ()
        raises (Reflective::MofError);
    void set_members (in Field new_value)
        raises (Reflective::MofError);
}; // end of interface StructDef

interface UnionDefClass : TypedefDefClass
{
    readonly attribute UnionDefSet all_of_type_union_def;
    readonly attribute UnionDefSet all_of_class_union_def;
    UnionDef create_union_def (
        in string identifier,
        in string repository_id,
        in string version,
        in UnionField union_members)
        raises (Reflective::MofError);
};

interface UnionDef : UnionDefClass, TypedefDef
{
    IDLType discriminator_type ()
        raises (Reflective::MofError);
    void set_discriminator_type (in IDLType new_value)
        raises (Reflective::MofError);
    UnionField union_members ()
```

```
        raises (Reflective::MofError);
    void set_union_members (in UnionField new_value)
        raises (Reflective::MofError);
}; // end of interface UnionDef

interface EnumDefClass : TypedefDefClass
{
    readonly attribute EnumDefSet all_of_type_enum_def;
    readonly attribute EnumDefSet all_of_class_enum_def;
    EnumDef create_enum_def (
        in string identifier,
        in string repository_id,
        in string version,
        in string members)
        raises (Reflective::MofError);
};

interface EnumDef : EnumDefClass, TypedefDef
{
    string members ()
        raises (Reflective::MofError);
    void set_members (in string new_value)
        raises (Reflective::MofError);
}; // end of interface EnumDef

interface AliasDefClass : TypedefDefClass, TypedClass
{
    readonly attribute AliasDefSet all_of_type_alias_def;
    readonly attribute AliasDefSet all_of_class_alias_def;
    AliasDef create_alias_def (
        in string identifier,
        in string repository_id,
        in string version)
        raises (Reflective::MofError);
};

interface AliasDef : AliasDefClass, TypedefDef, Typed
{
}; // end of interface AliasDef

interface StringDefClass : IDLTypeClass
{
    readonly attribute StringDefSet all_of_type_string_def;
    readonly attribute StringDefSet all_of_class_string_def;
    StringDef create_string_def (
        in unsigned long bound)
        raises (Reflective::MofError);
};

interface StringDef : StringDefClass, IDLType
{
```

```
  unsigned long bound ()
    raises (Reflective::MofError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::MofError);
}; // end of interface StringDef

interface WstringDefClass : IDLTypeClass
{
  readonly attribute WstringDefSet all_of_type_wstring_def;
  readonly attribute WstringDefSet all_of_class_wstring_def;
  WstringDef create_wstring_def (
    in unsigned long bound)
    raises (Reflective::MofError);
};

interface WstringDef : WstringDefClass, IDLType
{
  unsigned long bound ()
    raises (Reflective::MofError);
  void set_bound (in unsigned long new_value)
    raises (Reflective::MofError);
}; // end of interface WstringDef

interface FixedDefClass : IDLTypeClass
{
  readonly attribute FixedDefSet all_of_type_fixed_def;
  readonly attribute FixedDefSet all_of_class_fixed_def;
  FixedDef create_fixed_def (
    in unsigned short digits,
    in short scale)
    raises (Reflective::MofError);
};

interface FixedDef : FixedDefClass, IDLType
{
  unsigned short digits ()
    raises (Reflective::MofError);
  void set_digits (in unsigned short new_value)
    raises (Reflective::MofError);
  short scale ()
    raises (Reflective::MofError);
  void set_scale (in short new_value)
    raises (Reflective::MofError);
}; // end of interface FixedDef

interface SequenceDefClass : TypedClass, IDLTypeClass
{
  readonly attribute SequenceDefSet all_of_type_sequence_def;
  readonly attribute SequenceDefSet all_of_class_sequence_def;
  SequenceDef create_sequence_def (
    in unsigned long bound)
```

```
        raises (Reflective::MofError);
};

interface SequenceDef : SequenceDefClass, Typed, IDLType
{
    unsigned long bound ()
        raises (Reflective::MofError);
    void set_bound (in unsigned long new_value)
        raises (Reflective::MofError);
}; // end of interface SequenceDef

interface ArrayDefClass : TypedClass, IDLTypeClass
{
    readonly attribute ArrayDefSet all_of_type_array_def;
    readonly attribute ArrayDefSet all_of_class_array_def;
    ArrayDef create_array_def (
        in unsigned long bound)
        raises (Reflective::MofError);
};

interface ArrayDef : ArrayDefClass, Typed, IDLType
{
    unsigned long bound ()
        raises (Reflective::MofError);
    void set_bound (in unsigned long new_value)
        raises (Reflective::MofError);
}; // end of interface ArrayDef

interface PrimitiveDefClass : IDLTypeClass
{
    readonly attribute PrimitiveDefSet all_of_type_primitive_def;
    readonly attribute PrimitiveDefSet all_of_class_primitive_def;
    PrimitiveDef create_primitive_def (
        in PrimitiveKind kind)
        raises (Reflective::MofError);
};

interface PrimitiveDef : PrimitiveDefClass, IDLType
{
    PrimitiveKind kind ()
        raises (Reflective::MofError);
    void set_kind (in PrimitiveKind new_value)
        raises (Reflective::MofError);
}; // end of interface PrimitiveDef

interface UnionFieldClass : TypedClass
{
    readonly attribute UnionFieldSet all_of_type_union_field;
    readonly attribute UnionFieldSet all_of_class_union_field;
    UnionField create_union_field (
        in string identifier,
```

```
        in any label)
        raises (Reflective::MofError);
};

interface UnionField : UnionFieldClass, Typed
{
    string identifier ()
        raises (Reflective::MofError);
    void set_identifier (in string new_value)
        raises (Reflective::MofError);
    any label ()
        raises (Reflective::MofError);
    void set_label (in any new_value)
        raises (Reflective::MofError);
}; // end of interface UnionField

interface ValueMemberDefClass : TypedClass, ContainedClass
{
    readonly attribute ValueMemberDefSet all_of_type_value_member_def;
    readonly attribute ValueMemberDefSet all_of_class_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::MofError);
};

interface ValueMemberDef : ValueMemberDefClass, Typed, Contained
{
    boolean is_public_member ()
        raises (Reflective::MofError);
    void set_is_public_member (in boolean new_value)
        raises (Reflective::MofError);
}; // end of interface ValueMemberDef

interface ValueDefClass : ContainerClass, IDLTypeClass
{
    readonly attribute ValueDefSet all_of_type_value_def;
    readonly attribute ValueDefSet all_of_class_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::MofError);
};

interface ValueDef : ValueDefClass, Container, IDLType
```

```
{
  InterfaceDef interface_def ()
    raises (Reflective::NotSet, Reflective::MofError);
  void set_interface_def (in InterfaceDef new_value)
    raises (Reflective::MofError);
  void unset_interface_def ()
    raises (Reflective::MofError);
  ValueDef base ()
    raises (Reflective::NotSet, Reflective::MofError);
  void set_base (in ValueDef new_value)
    raises (Reflective::MofError);
  void unset_base ()
    raises (Reflective::MofError);
  ValueDefSet abstract_base ()
    raises (Reflective::MofError);
  void set_abstract_base (in ValueDefSet new_value)
    raises (Reflective::MofError);
  void unset_abstract_base ()
    raises (Reflective::MofError);
  void add_abstract_base (in ValueDef new_element)
    raises (Reflective::MofError);
  void modify_abstract_base (
    in ValueDef old_element,
    in ValueDef new_element)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove_abstract_base (in ValueDef old_element)
    raises (Reflective::NotFound, Reflective::MofError);
  boolean is_abstract ()
    raises (Reflective::MofError);
  void set_is_abstract (in boolean new_value)
    raises (Reflective::MofError);
  boolean is_custom ()
    raises (Reflective::MofError);
  void set_is_custom (in boolean new_value)
    raises (Reflective::MofError);
  boolean is_truncatable ()
    raises (Reflective::MofError);
  void set_is_truncatable (in boolean new_value)
    raises (Reflective::MofError);
}; // end of interface ValueDef

interface ValueBoxDefClass : TypedefDefClass
{
  readonly attribute ValueBoxDefSet all_of_type_value_box_def;
  readonly attribute ValueBoxDefSet all_of_class_value_box_def;
  ValueBoxDef create_value_box_def (
    in string identifier,
    in string repository_id,
    in string version)
    raises (Reflective::MofError);
};
```

```
interface ValueBoxDef : ValueBoxDefClass, TypedefDef
{
}; // end of interface ValueBoxDef

interface OperationDefClass : TypedClass, ContainedClass
{
  readonly attribute OperationDefSet all_of_type_operation_def;
  readonly attribute OperationDefSet all_of_class_operation_def;
  OperationDef create_operation_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_oneway,
    in ParameterDef parameters,
    in string contexts)
    raises (Reflective::MofError);
};

interface OperationDef : OperationDefClass, Typed, Contained
{
  ExceptionDefSet exception_def ()
    raises (Reflective::MofError);
  void set_exception_def (in ExceptionDefSet new_value)
    raises (Reflective::MofError);
  void unset_exception_def ()
    raises (Reflective::MofError);
  void add_exception_def (in ExceptionDef new_element)
    raises (Reflective::MofError);
  void modify_exception_def (
    in ExceptionDef old_element,
    in ExceptionDef new_element)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove_exception_def (in ExceptionDef old_element)
    raises (Reflective::NotFound, Reflective::MofError);
  boolean is_oneway ()
    raises (Reflective::MofError);
  void set_is_oneway (in boolean new_value)
    raises (Reflective::MofError);
  ParameterDef parameters ()
    raises (Reflective::MofError);
  void set_parameters (in ParameterDef new_value)
    raises (Reflective::MofError);
  string contexts ()
    raises (Reflective::MofError);
  void set_contexts (in string new_value)
    raises (Reflective::MofError);
}; // end of interface OperationDef

interface ExceptionDefClass : ContainedClass
{
```

```
  readonly attribute ExceptionDefSet all_of_type_exception_def;
  readonly attribute ExceptionDefSet all_of_class_exception_def;
  ExceptionDef create_exception_def (
    in string identifier,
    in string repository_id,
    in string version,
    in CORBA::TypeCode type_code,
    in Field members)
    raises (Reflective::MofError);
};

interface ExceptionDef : ExceptionDefClass, Contained
{
  CORBA::TypeCode type_code ()
    raises (Reflective::MofError);
  Field members ()
    raises (Reflective::MofError);
  void set_members (in Field new_value)
    raises (Reflective::MofError);
}; // end of interface ExceptionDef

interface AttributeDefClass : TypedClass, ContainedClass
{
  readonly attribute AttributeDefSet all_of_type_attribute_def;
  readonly attribute AttributeDefSet all_of_class_attribute_def;
  AttributeDef create_attribute_def (
    in string identifier,
    in string repository_id,
    in string version,
    in boolean is_readonly)
    raises (Reflective::MofError);
};

interface AttributeDef : AttributeDefClass, Typed, Contained
{
  ExceptionDefSet set_exception ()
    raises (Reflective::MofError);
  void set_set_exception (in ExceptionDefSet new_value)
    raises (Reflective::MofError);
  void unset_set_exception ()
    raises (Reflective::MofError);
  void add_set_exception (in ExceptionDef new_element)
    raises (Reflective::MofError);
  void modify_set_exception (
    in ExceptionDef old_element,
    in ExceptionDef new_element)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove_set_exception (in ExceptionDef old_element)
    raises (Reflective::NotFound, Reflective::MofError);
  ExceptionDefSet get_exception ()
    raises (Reflective::MofError);
```

```
    void set_get_exception (in ExceptionDefSet new_value)
      raises (Reflective::MofError);
    void unset_get_exception ()
      raises (Reflective::MofError);
    void add_get_exception (in ExceptionDef new_element)
      raises (Reflective::MofError);
    void modify_get_exception (
      in ExceptionDef old_element,
      in ExceptionDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_get_exception (in ExceptionDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    boolean is_readonly ()
      raises (Reflective::MofError);
    void set_is_readonly (in boolean new_value)
      raises (Reflective::MofError);
  }; // end of interface AttributeDef

  struct InterfaceDerivedFromLink
  {
    InterfaceDef base;
    InterfaceDef derived;
  };
  typedef sequence<InterfaceDerivedFromLink> InterfaceDerivedFromLink-
Set;

  interface InterfaceDerivedFrom : Reflective::RefAssociation
  {
    InterfaceDerivedFromLinkSet all_interface_derived_from_links()
      raises (Reflective::MofError);
    boolean exists (
      in InterfaceDef base,
      in InterfaceDef derived)
      raises (Reflective::MofError);
    InterfaceDefSet base (in InterfaceDef derived)
      raises (Reflective::MofError);
    void add (
      in InterfaceDef base,
      in InterfaceDef derived)
      raises (Reflective::MofError);
    void modify_base (
      in InterfaceDef base,
      in InterfaceDef derived,
      in InterfaceDef new_base)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in InterfaceDef base,
      in InterfaceDef derived)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // 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()
    raises (Reflective::MofError);
  boolean exists (
    in IDLType discriminator_type,
    in UnionDef union_def)
    raises (Reflective::MofError);
  IDLType discriminator_type (in UnionDef union_def)
    raises (Reflective::MofError);
  void add (
    in IDLType discriminator_type,
    in UnionDef union_def)
    raises (Reflective::MofError);
  void modify_discriminator_type (
    in IDLType discriminator_type,
    in UnionDef union_def,
    in IDLType new_discriminator_type)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in IDLType discriminator_type,
    in UnionDef union_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface DiscriminatedBy

struct TypedByLink
{
  IDLType idl_type;
  BaseIDL::Typed typed;
};
typedef sequence<TypedByLink> TypedByLinkSet;

interface TypedBy : Reflective::RefAssociation
{
  TypedByLinkSet all_typed_by_links()
    raises (Reflective::MofError);
  boolean exists (
    in IDLType idl_type,
    in BaseIDL::Typed typed)
    raises (Reflective::MofError);
  IDLType idl_type (in BaseIDL::Typed typed)
    raises (Reflective::MofError);
  void add (
    in IDLType idl_type,
```

```
    in BaseIDL::Typed typed)
    raises (Reflective::MofError);
  void modify_idl_type (
    in IDLType idl_type,
    in BaseIDL::Typed typed,
    in IDLType new_idl_type)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in IDLType idl_type,
    in BaseIDL::Typed typed)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface TypedBy

struct SupportsLink
{
  InterfaceDef interface_def;
  ValueDef value_def;
};
typedef sequence<SupportsLink> SupportsLinkSet;

interface Supports : Reflective::RefAssociation
{
  SupportsLinkSet all_supports_links()
    raises (Reflective::MofError);
  boolean exists (
    in InterfaceDef interface_def,
    in ValueDef value_def)
    raises (Reflective::MofError);
  InterfaceDef interface_def (in ValueDef value_def)
    raises (Reflective::MofError);
  void add (
    in InterfaceDef interface_def,
    in ValueDef value_def)
    raises (Reflective::MofError);
  void modify_interface_def (
    in InterfaceDef interface_def,
    in ValueDef value_def,
    in InterfaceDef new_interface_def)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in InterfaceDef interface_def,
    in ValueDef value_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface Supports

struct ValueDerivedFromLink
{
  ValueDef base;
  ValueDef derived;
};
typedef sequence<ValueDerivedFromLink> ValueDerivedFromLinkSet;
```

```
  interface ValueDerivedFrom : Reflective::RefAssociation
  {
    ValueDerivedFromLinkSet all_value_derived_from_links()
      raises (Reflective::MofError);
    boolean exists (
      in ValueDef base,
      in ValueDef derived)
      raises (Reflective::MofError);
    ValueDef base (in ValueDef derived)
      raises (Reflective::MofError);
    void add (
      in ValueDef base,
      in ValueDef derived)
      raises (Reflective::MofError);
    void modify_base (
      in ValueDef base,
      in ValueDef derived,
      in ValueDef new_base)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ValueDef base,
      in ValueDef derived)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface ValueDerivedFrom

  struct AbstractDerivedFromLink
  {
    ValueDef abstract_derived;
    ValueDef abstract_base;
  };
  typedef sequence<AbstractDerivedFromLink> AbstractDerivedFromLink-
Set;

  interface AbstractDerivedFrom : Reflective::RefAssociation
  {
    AbstractDerivedFromLinkSet all_abstract_derived_from_links()
      raises (Reflective::MofError);
    boolean exists (
      in ValueDef abstract_derived,
      in ValueDef abstract_base)
      raises (Reflective::MofError);
    ValueDefSet abstract_base (in ValueDef abstract_derived)
      raises (Reflective::MofError);
    void add (
      in ValueDef abstract_derived,
      in ValueDef abstract_base)
      raises (Reflective::MofError);
    void modify_abstract_base (
      in ValueDef abstract_derived,
      in ValueDef abstract_base,
```

```
    in ValueDef new_abstract_base)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ValueDef abstract_derived,
    in ValueDef abstract_base)
    raises (Reflective::NotFound, Reflective::MofError);
}; // 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()
    raises (Reflective::MofError);
  boolean exists (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute)
    raises (Reflective::MofError);
  ExceptionDefSet set_exception (in AttributeDef set_attribute)
    raises (Reflective::MofError);
  void add (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute)
    raises (Reflective::MofError);
  void modify_set_exception (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute,
    in ExceptionDef new_set_exception)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ExceptionDef set_exception,
    in AttributeDef set_attribute)
    raises (Reflective::NotFound, Reflective::MofError);
}; // 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()
    raises (Reflective::MofError);
```

```
  boolean exists (
    in ExceptionDef exception_def,
    in OperationDef operation_def)
    raises (Reflective::MofError);
  ExceptionDefSet exception_def (in OperationDef operation_def)
    raises (Reflective::MofError);
  void add (
    in ExceptionDef exception_def,
    in OperationDef operation_def)
    raises (Reflective::MofError);
  void modify_exception_def (
    in ExceptionDef exception_def,
    in OperationDef operation_def,
    in ExceptionDef new_exception_def)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ExceptionDef exception_def,
    in OperationDef operation_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // 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()
    raises (Reflective::MofError);
  boolean exists (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute)
    raises (Reflective::MofError);
  ExceptionDefSet get_exception (in AttributeDef get_attribute)
    raises (Reflective::MofError);
  void add (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute)
    raises (Reflective::MofError);
  void modify_get_exception (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute,
    in ExceptionDef new_get_exception)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ExceptionDef get_exception,
    in AttributeDef get_attribute)
    raises (Reflective::NotFound, Reflective::MofError);
```

```
}; // end of interface GetRaises

struct ContainsLink
{
  Container defined_in;
  Contained contents;
};
typedef sequence<ContainsLink> ContainsLinkSet;

interface Contains : Reflective::RefAssociation
{
  ContainsLinkSet all_contains_links()
    raises (Reflective::MofError);
  boolean exists (
    in Container defined_in,
    in Contained contents)
    raises (Reflective::MofError);
  Container defined_in (in Contained contents)
    raises (Reflective::MofError);
  ContainedSet contents (in Container defined_in)
    raises (Reflective::MofError);
  void add (
    in Container defined_in,
    in Contained contents)
    raises (Reflective::MofError);
  void modify_defined_in (
    in Container defined_in,
    in Contained contents,
    in Container new_defined_in)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_contents (
    in Container defined_in,
    in Contained contents,
    in Contained new_contents)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in Container defined_in,
    in Contained contents)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface Contains

interface BaseIDLPackageFactory
{
  BaseIDLPackage create_base_idl_package ()
    raises (Reflective::MofError);
};

interface BaseIDLPackage : Reflective::RefPackage
{
  readonly attribute TypedClass typed_ref;
  readonly attribute ParameterDefClass parameter_def_ref;
```

```
        readonly attribute ContainedClass contained_ref;
        readonly attribute ConstantDefClass constant_def_ref;
        readonly attribute ContainerClass container_ref;
        readonly attribute ModuleDefClass module_def_ref;
        readonly attribute IDLTypeClass idltype_ref;
        readonly attribute TypedefDefClass typedef_def_ref;
        readonly attribute InterfaceDefClass interface_def_ref;
        readonly attribute FieldClass field_ref;
        readonly attribute StructDefClass struct_def_ref;
        readonly attribute UnionDefClass union_def_ref;
        readonly attribute EnumDefClass enum_def_ref;
        readonly attribute AliasDefClass alias_def_ref;
        readonly attribute StringDefClass string_def_ref;
        readonly attribute WstringDefClass wstring_def_ref;
        readonly attribute FixedDefClass fixed_def_ref;
        readonly attribute SequenceDefClass sequence_def_ref;
        readonly attribute ArrayDefClass array_def_ref;
        readonly attribute PrimitiveDefClass primitive_def_ref;
        readonly attribute UnionFieldClass union_field_ref;
        readonly attribute ValueMemberDefClass value_member_def_ref;
        readonly attribute ValueDefClass value_def_ref;
        readonly attribute ValueBoxDefClass value_box_def_ref;
        readonly attribute OperationDefClass operation_def_ref;
        readonly attribute ExceptionDefClass exception_def_ref;
        readonly attribute AttributeDefClass attribute_def_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;
    };
}; // end of module BaseIDL
```

8.3.3 IDL for the ComponentIDL Package

```
#pragma prefix "ccm.omg.org"
#include "BaseIDL.idl"

module ComponentIDL
{
    interface ComponentFeatureClass;
    interface ComponentFeature;
    typedef sequence<ComponentFeature> ComponentFeatureSet;
    interface ComponentDefClass;
```

```
   interface ComponentDef;
   typedef sequence<ComponentDef> ComponentDefSet;
   interface ProvidesDefClass;
   interface ProvidesDef;
   typedef sequence<ProvidesDef> ProvidesDefSet;
   interface HomeDefClass;
   interface HomeDef;
   typedef sequence<HomeDef> HomeDefSet;
   interface FactoryDefClass;
   interface FactoryDef;
   typedef sequence<FactoryDef> FactoryDefSet;
   interface FinderDefClass;
   interface FinderDef;
   typedef sequence<FinderDef> FinderDefSet;
   interface EventPortDefClass;
   interface EventPortDef;
   typedef sequence<EventPortDef> EventPortDefSet;
   interface EmitsDefClass;
   interface EmitsDef;
   typedef sequence<EmitsDef> EmitsDefSet;
   interface ConsumesDefClass;
   interface ConsumesDef;
   typedef sequence<ConsumesDef> ConsumesDefSet;
   interface UsesDefClass;
   interface UsesDef;
   typedef sequence<UsesDef> UsesDefSet;
   interface PublishesDefClass;
   interface PublishesDef;
   typedef sequence<PublishesDef> PublishesDefSet;
   interface EventDefClass;
   interface EventDef;
   typedef sequence<EventDef> EventDefSet;
   interface ComponentIDLPackage;

   interface ComponentFeatureClass : Reflective::RefObject
   {
      readonly attribute ComponentFeatureSet
all_of_type_component_feature;
   };

   interface ComponentFeature : ComponentFeatureClass
   {
   }; // end of interface ComponentFeature

   interface ComponentDefClass : BaseIDL::InterfaceDefClass, Component-
FeatureClass
   {
      readonly attribute ComponentDefSet all_of_type_component_def;
      readonly attribute ComponentDefSet all_of_class_component_def;
      ComponentDef create_component_def (
         in string identifier,
```

```
        in string repository_id,
        in string version,
        in boolean is_abstract,
        in boolean is_local)
        raises (Reflective::MofError);
  };

  interface ComponentDef : ComponentDefClass, BaseIDL::InterfaceDef,
ComponentFeature
  {
    ProvidesDefSet facet ()
      raises (Reflective::MofError);
    void set_facet (in ProvidesDefSet new_value)
      raises (Reflective::MofError);
    void unset_facet ()
      raises (Reflective::MofError);
    void add_facet (in ProvidesDef new_element)
      raises (Reflective::MofError);
    void modify_facet (
      in ProvidesDef old_element,
      in ProvidesDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_facet (in ProvidesDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    EmitsDefSet emits ()
      raises (Reflective::MofError);
    void set_emits (in EmitsDefSet new_value)
      raises (Reflective::MofError);
    void unset_emits ()
      raises (Reflective::MofError);
    void add_emits (in EmitsDef new_element)
      raises (Reflective::MofError);
    void modify_emits (
      in EmitsDef old_element,
      in EmitsDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_emits (in EmitsDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    ConsumesDefSet consumes ()
      raises (Reflective::MofError);
    void set_consumes (in ConsumesDefSet new_value)
      raises (Reflective::MofError);
    void unset_consumes ()
      raises (Reflective::MofError);
    void add_consumes (in ConsumesDef new_element)
      raises (Reflective::MofError);
    void modify_consumes (
      in ConsumesDef old_element,
      in ConsumesDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_consumes (in ConsumesDef old_element)
```

```
      raises (Reflective::NotFound, Reflective::MofError);
    UsesDefSet receptacle ()
      raises (Reflective::MofError);
    void set_receptacle (in UsesDefSet new_value)
      raises (Reflective::MofError);
    void unset_receptacle ()
      raises (Reflective::MofError);
    void add_receptacle (in UsesDef new_element)
      raises (Reflective::MofError);
    void modify_receptacle (
      in UsesDef old_element,
      in UsesDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_receptacle (in UsesDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    BaseIDL::InterfaceDefSet supports ()
      raises (Reflective::MofError);
    void set_supports (in BaseIDL::InterfaceDefSet new_value)
      raises (Reflective::MofError);
    void unset_supports ()
      raises (Reflective::MofError);
    void add_supports (in BaseIDL::InterfaceDef new_element)
      raises (Reflective::MofError);
    void modify_supports (
      in BaseIDL::InterfaceDef old_element,
      in BaseIDL::InterfaceDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_supports (in BaseIDL::InterfaceDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    PublishesDefSet publishes ()
      raises (Reflective::MofError);
    void set_publishes (in PublishesDefSet new_value)
      raises (Reflective::MofError);
    void unset_publishes ()
      raises (Reflective::MofError);
    void add_publishes (in PublishesDef new_element)
      raises (Reflective::MofError);
    void modify_publishes (
      in PublishesDef old_element,
      in PublishesDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_publishes (in PublishesDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface ComponentDef

  interface ProvidesDefClass : BaseIDL::ContainedClass, ComponentFea-
tureClass
  {
    readonly attribute ProvidesDefSet all_of_type_provides_def;
    readonly attribute ProvidesDefSet all_of_class_provides_def;
    ProvidesDef create_provides_def (
```

```
      in string identifier,
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface ProvidesDef : ProvidesDefClass, BaseIDL::Contained, ComponentFeature
  {
    BaseIDL::InterfaceDef provides ()
      raises (Reflective::MofError);
    void set_provides (in BaseIDL::InterfaceDef new_value)
      raises (Reflective::MofError);
    ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
      raises (Reflective::MofError);
  }; // end of interface ProvidesDef

  interface HomeDefClass : BaseIDL::InterfaceDefClass
  {
    readonly attribute HomeDefSet all_of_type_home_def;
    readonly attribute HomeDefSet all_of_class_home_def;
    HomeDef create_home_def (
      in string identifier,
      in string repository_id,
      in string version,
      in boolean is_abstract,
      in boolean is_local)
      raises (Reflective::MofError);
  };

  interface HomeDef : HomeDefClass, BaseIDL::InterfaceDef
  {
    FinderDefSet finder ()
      raises (Reflective::MofError);
    void set_finder (in FinderDefSet new_value)
      raises (Reflective::MofError);
    void unset_finder ()
      raises (Reflective::MofError);
    void add_finder (in FinderDef new_element)
      raises (Reflective::MofError);
    void modify_finder (
      in FinderDef old_element,
      in FinderDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_finder (in FinderDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
```

```
        raises (Reflective::MofError);
    FactoryDefSet factory ()
        raises (Reflective::MofError);
    void set_factory (in FactoryDefSet new_value)
        raises (Reflective::MofError);
    void unset_factory ()
        raises (Reflective::MofError);
    void add_factory (in FactoryDef new_element)
        raises (Reflective::MofError);
    void modify_factory (
        in FactoryDef old_element,
        in FactoryDef new_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove_factory (in FactoryDef old_element)
        raises (Reflective::NotFound, Reflective::MofError);
    BaseIDL::ValueDef primary_key ()
        raises (Reflective::NotSet, Reflective::MofError);
    void set_primary_key (in BaseIDL::ValueDef new_value)
        raises (Reflective::MofError);
    void unset_primary_key ()
        raises (Reflective::MofError);
    BaseIDL::InterfaceDefSet supports ()
        raises (Reflective::MofError);
    void set_supports (in BaseIDL::InterfaceDefSet new_value)
        raises (Reflective::MofError);
    void unset_supports ()
        raises (Reflective::MofError);
    void add_supports (in BaseIDL::InterfaceDef new_element)
        raises (Reflective::MofError);
    void modify_supports (
        in BaseIDL::InterfaceDef old_element,
        in BaseIDL::InterfaceDef new_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove_supports (in BaseIDL::InterfaceDef old_element)
        raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface HomeDef

interface FactoryDefClass : BaseIDL::OperationDefClass
{
    readonly attribute FactoryDefSet all_of_type_factory_def;
    readonly attribute FactoryDefSet all_of_class_factory_def;
    FactoryDef create_factory_def (
        in string identifier,
        in string repository_id,
        in string version,
        in boolean is_oneway,
        in BaseIDL::ParameterDef parameters,
        in string contexts)
        raises (Reflective::MofError);
};
```

```
  interface FactoryDef : FactoryDefClass, BaseIDL::OperationDef
  {
    HomeDef home ()
      raises (Reflective::MofError);
    void set_home (in HomeDef new_value)
      raises (Reflective::MofError);
  }; // end of interface FactoryDef

  interface FinderDefClass : BaseIDL::OperationDefClass
  {
    readonly attribute FinderDefSet all_of_type_finder_def;
    readonly attribute FinderDefSet all_of_class_finder_def;
    FinderDef create_finder_def (
      in string identifier,
      in string repository_id,
      in string version,
      in boolean is_oneway,
      in BaseIDL::ParameterDef parameters,
      in string contexts)
      raises (Reflective::MofError);
  };

  interface FinderDef : FinderDefClass, BaseIDL::OperationDef
  {
    HomeDef home ()
      raises (Reflective::MofError);
    void set_home (in HomeDef new_value)
      raises (Reflective::MofError);
  }; // end of interface FinderDef

  interface EventPortDefClass : BaseIDL::ContainedClass, ComponentFea-
tureClass
  {
    readonly attribute EventPortDefSet all_of_type_event_port_def;
  };

  interface EventPortDef : EventPortDefClass, BaseIDL::Contained, Compo-
nentFeature
  {
    EventDef type ()
      raises (Reflective::MofError);
    void set_type (in EventDef new_value)
      raises (Reflective::MofError);
  }; // end of interface EventPortDef

  interface EmitsDefClass : EventPortDefClass, ComponentFeatureClass
  {
    readonly attribute EmitsDefSet all_of_type_emits_def;
    readonly attribute EmitsDefSet all_of_class_emits_def;
    EmitsDef create_emits_def (
      in string identifier,
```

```
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface EmitsDef : EmitsDefClass, EventPortDef, ComponentFeature
  {
    ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
      raises (Reflective::MofError);
  }; // end of interface EmitsDef

  interface ConsumesDefClass : EventPortDefClass, ComponentFeature-
Class
  {
    readonly attribute ConsumesDefSet all_of_type_consumes_def;
    readonly attribute ConsumesDefSet all_of_class_consumes_def;
    ConsumesDef create_consumes_def (
      in string identifier,
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface ConsumesDef : ConsumesDefClass, EventPortDef, Component-
Feature
  {
    ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
      raises (Reflective::MofError);
  }; // end of interface ConsumesDef

  interface UsesDefClass : BaseIDL::ContainedClass, ComponentFeature-
Class
  {
    readonly attribute UsesDefSet all_of_type_uses_def;
    readonly attribute UsesDefSet all_of_class_uses_def;
    UsesDef create_uses_def (
      in string identifier,
      in string repository_id,
      in string version,
      in boolean multiple)
      raises (Reflective::MofError);
  };

  interface UsesDef : UsesDefClass, BaseIDL::Contained, ComponentFea-
ture
  {
    ComponentDef component ()
```

```
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
      raises (Reflective::MofError);
    BaseIDL::InterfaceDef uses ()
      raises (Reflective::MofError);
    void set_uses (in BaseIDL::InterfaceDef new_value)
      raises (Reflective::MofError);
    boolean multiple ()
      raises (Reflective::MofError);
    void set_multiple (in boolean new_value)
      raises (Reflective::MofError);
  }; // end of interface UsesDef

  interface PublishesDefClass : EventPortDefClass, ComponentFeatureClass
  {
    readonly attribute PublishesDefSet all_of_type_publishes_def;
    readonly attribute PublishesDefSet all_of_class_publishes_def;
    PublishesDef create_publishes_def (
      in string identifier,
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface PublishesDef : PublishesDefClass, EventPortDef, ComponentFeature
  {
    ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentDef new_value)
      raises (Reflective::MofError);
  }; // end of interface PublishesDef

  interface EventDefClass : BaseIDL::ValueDefClass
  {
    readonly attribute EventDefSet all_of_type_event_def;
    readonly attribute EventDefSet all_of_class_event_def;
    EventDef create_event_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::MofError);
  };

  interface EventDef : EventDefClass, BaseIDL::ValueDef
  {
  }; // end of interface EventDef
```

```
struct ProvidesInterfaceLink
{
  BaseIDL::InterfaceDef provides;
  ProvidesDef provides_def;
};
typedef sequence<ProvidesInterfaceLink> ProvidesInterfaceLinkSet;

interface ProvidesInterface : Reflective::RefAssociation
{
  ProvidesInterfaceLinkSet all_provides_interface_links()
    raises (Reflective::MofError);
  boolean exists (
    in BaseIDL::InterfaceDef provides,
    in ProvidesDef provides_def)
    raises (Reflective::MofError);
  BaseIDL::InterfaceDef provides (in ProvidesDef provides_def)
    raises (Reflective::MofError);
  void add (
    in BaseIDL::InterfaceDef provides,
    in ProvidesDef provides_def)
    raises (Reflective::MofError);
  void modify_provides (
    in BaseIDL::InterfaceDef provides,
    in ProvidesDef provides_def,
    in BaseIDL::InterfaceDef new_provides)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in BaseIDL::InterfaceDef provides,
    in ProvidesDef provides_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ProvidesInterface

struct ComponentFacetLink
{
  ComponentDef component;
  ProvidesDef facet;
};
typedef sequence<ComponentFacetLink> ComponentFacetLinkSet;

interface ComponentFacet : Reflective::RefAssociation
{
  ComponentFacetLinkSet all_component_facet_links()
    raises (Reflective::MofError);
  boolean exists (
    in ComponentDef component,
    in ProvidesDef facet)
    raises (Reflective::MofError);
  ComponentDef component (in ProvidesDef facet)
    raises (Reflective::MofError);
  ProvidesDefSet facet (in ComponentDef component)
```

```
    raises (Reflective::MofError);
  void add (
    in ComponentDef component,
    in ProvidesDef facet)
    raises (Reflective::MofError);
  void modify_component (
    in ComponentDef component,
    in ProvidesDef facet,
    in ComponentDef new_component)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_facet (
    in ComponentDef component,
    in ProvidesDef facet,
    in ProvidesDef new_facet)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ComponentDef component,
    in ProvidesDef facet)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ComponentFacet

struct HomeFinderLink
{
  HomeDef home;
  FinderDef finder;
};
typedef sequence<HomeFinderLink> HomeFinderLinkSet;

interface HomeFinder : Reflective::RefAssociation
{
  HomeFinderLinkSet all_home_finder_links()
    raises (Reflective::MofError);
  boolean exists (
    in HomeDef home,
    in FinderDef finder)
    raises (Reflective::MofError);
  HomeDef home (in FinderDef finder)
    raises (Reflective::MofError);
  FinderDefSet finder (in HomeDef home)
    raises (Reflective::MofError);
  void add (
    in HomeDef home,
    in FinderDef finder)
    raises (Reflective::MofError);
  void modify_home (
    in HomeDef home,
    in FinderDef finder,
    in HomeDef new_home)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_finder (
    in HomeDef home,
```

```
    in FinderDef finder,
    in FinderDef new_finder)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in HomeDef home,
    in FinderDef finder)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface HomeFinder

struct ComponentEmitsLink
{
  ComponentDef component;
  EmitsDef emits;
};
typedef sequence<ComponentEmitsLink> ComponentEmitsLinkSet;

interface ComponentEmits : Reflective::RefAssociation
{
  ComponentEmitsLinkSet all_component_emits_links()
    raises (Reflective::MofError);
  boolean exists (
    in ComponentDef component,
    in EmitsDef emits)
    raises (Reflective::MofError);
  ComponentDef component (in EmitsDef emits)
    raises (Reflective::MofError);
  EmitsDefSet emits (in ComponentDef component)
    raises (Reflective::MofError);
  void add (
    in ComponentDef component,
    in EmitsDef emits)
    raises (Reflective::MofError);
  void modify_component (
    in ComponentDef component,
    in EmitsDef emits,
    in ComponentDef new_component)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_emits (
    in ComponentDef component,
    in EmitsDef emits,
    in EmitsDef new_emits)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ComponentDef component,
    in EmitsDef emits)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ComponentEmits

struct ComponentConsumesLink
{
  ComponentDef component;
```

```
    ConsumesDef consumes;
  };
  typedef sequence<ComponentConsumesLink> ComponentCon-
sumesLinkSet;

  interface ComponentConsumes : Reflective::RefAssociation
  {
    ComponentConsumesLinkSet all_component_consumes_links()
      raises (Reflective::MofError);
    boolean exists (
      in ComponentDef component,
      in ConsumesDef consumes)
      raises (Reflective::MofError);
    ComponentDef component (in ConsumesDef consumes)
      raises (Reflective::MofError);
    ConsumesDefSet consumes (in ComponentDef component)
      raises (Reflective::MofError);
    void add (
      in ComponentDef component,
      in ConsumesDef consumes)
      raises (Reflective::MofError);
    void modify_component (
      in ComponentDef component,
      in ConsumesDef consumes,
      in ComponentDef new_component)
      raises (Reflective::NotFound, Reflective::MofError);
    void modify_consumes (
      in ComponentDef component,
      in ConsumesDef consumes,
      in ConsumesDef new_consumes)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ComponentDef component,
      in ConsumesDef consumes)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface ComponentConsumes

  struct ComponentReceptacleLink
  {
    ComponentDef component;
    UsesDef receptacle;
  };
  typedef sequence<ComponentReceptacleLink> ComponentRecepta-
cleLinkSet;

  interface ComponentReceptacle : Reflective::RefAssociation
  {
    ComponentReceptacleLinkSet all_component_receptacle_links()
      raises (Reflective::MofError);
    boolean exists (
      in ComponentDef component,
```

```
    in UsesDef receptacle)
    raises (Reflective::MofError);
  ComponentDef component (in UsesDef receptacle)
    raises (Reflective::MofError);
  UsesDefSet receptacle (in ComponentDef component)
    raises (Reflective::MofError);
  void add (
    in ComponentDef component,
    in UsesDef receptacle)
    raises (Reflective::MofError);
  void modify_component (
    in ComponentDef component,
    in UsesDef receptacle,
    in ComponentDef new_component)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_receptacle (
    in ComponentDef component,
    in UsesDef receptacle,
    in UsesDef new_receptacle)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in ComponentDef component,
    in UsesDef receptacle)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ComponentReceptacle

struct UsesInterfaceLink
{
  BaseIDL::InterfaceDef uses;
  UsesDef uses_def;
};
typedef sequence<UsesInterfaceLink> UsesInterfaceLinkSet;

interface UsesInterface : Reflective::RefAssociation
{
  UsesInterfaceLinkSet all_uses_interface_links()
    raises (Reflective::MofError);
  boolean exists (
    in BaseIDL::InterfaceDef uses,
    in UsesDef uses_def)
    raises (Reflective::MofError);
  BaseIDL::InterfaceDef uses (in UsesDef uses_def)
    raises (Reflective::MofError);
  void add (
    in BaseIDL::InterfaceDef uses,
    in UsesDef uses_def)
    raises (Reflective::MofError);
  void modify_uses (
    in BaseIDL::InterfaceDef uses,
    in UsesDef uses_def,
    in BaseIDL::InterfaceDef new_uses)
```

```
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in BaseIDL::InterfaceDef uses,
      in UsesDef uses_def)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface UsesInterface

  struct ComponentHomeLink
  {
    ComponentDef component;
    HomeDef home;
  };
  typedef sequence<ComponentHomeLink> ComponentHomeLinkSet;

  interface ComponentHome : Reflective::RefAssociation
  {
    ComponentHomeLinkSet all_component_home_links()
      raises (Reflective::MofError);
    boolean exists (
      in ComponentDef component,
      in HomeDef home)
      raises (Reflective::MofError);
    ComponentDef component (in HomeDef home)
      raises (Reflective::MofError);
    void add (
      in ComponentDef component,
      in HomeDef home)
      raises (Reflective::MofError);
    void modify_component (
      in ComponentDef component,
      in HomeDef home,
      in ComponentDef new_component)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ComponentDef component,
      in HomeDef home)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface ComponentHome

  struct ComponentSupportsLink
  {
    BaseIDL::InterfaceDef supports;
    ComponentDef components;
  };
  typedef sequence<ComponentSupportsLink> ComponentSupportsLink-
Set;

  interface ComponentSupports : Reflective::RefAssociation
  {
    ComponentSupportsLinkSet all_component_supports_links()
      raises (Reflective::MofError);
```

```
  boolean exists (
    in BaseIDL::InterfaceDef supports,
    in ComponentDef components)
    raises (Reflective::MofError);
  BaseIDL::InterfaceDefSet supports (in ComponentDef components)
    raises (Reflective::MofError);
  void add (
    in BaseIDL::InterfaceDef supports,
    in ComponentDef components)
    raises (Reflective::MofError);
  void modify_supports (
    in BaseIDL::InterfaceDef supports,
    in ComponentDef components,
    in BaseIDL::InterfaceDef new_supports)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in BaseIDL::InterfaceDef supports,
    in ComponentDef components)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ComponentSupports

struct HomeFactoryLink
{
  HomeDef home;
  FactoryDef factory;
};
typedef sequence<HomeFactoryLink> HomeFactoryLinkSet;

interface HomeFactory : Reflective::RefAssociation
{
  HomeFactoryLinkSet all_home_factory_links()
    raises (Reflective::MofError);
  boolean exists (
    in HomeDef home,
    in FactoryDef factory)
    raises (Reflective::MofError);
  HomeDef home (in FactoryDef factory)
    raises (Reflective::MofError);
  FactoryDefSet factory (in HomeDef home)
    raises (Reflective::MofError);
  void add (
    in HomeDef home,
    in FactoryDef factory)
    raises (Reflective::MofError);
  void modify_home (
    in HomeDef home,
    in FactoryDef factory,
    in HomeDef new_home)
    raises (Reflective::NotFound, Reflective::MofError);
  void modify_factory (
    in HomeDef home,
```

```
      in FactoryDef factory,
      in FactoryDef new_factory)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in HomeDef home,
      in FactoryDef factory)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface HomeFactory

  struct ComponentPublishesLink
  {
    ComponentDef component;
    PublishesDef publishes;
  };
  typedef sequence<ComponentPublishesLink> ComponentPublishesLink-
Set;

  interface ComponentPublishes : Reflective::RefAssociation
  {
    ComponentPublishesLinkSet all_component_publishes_links()
      raises (Reflective::MofError);
    boolean exists (
      in ComponentDef component,
      in PublishesDef publishes)
      raises (Reflective::MofError);
    ComponentDef component (in PublishesDef publishes)
      raises (Reflective::MofError);
    PublishesDefSet publishes (in ComponentDef component)
      raises (Reflective::MofError);
    void add (
      in ComponentDef component,
      in PublishesDef publishes)
      raises (Reflective::MofError);
    void modify_component (
      in ComponentDef component,
      in PublishesDef publishes,
      in ComponentDef new_component)
      raises (Reflective::NotFound, Reflective::MofError);
    void modify_publishes (
      in ComponentDef component,
      in PublishesDef publishes,
      in PublishesDef new_publishes)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ComponentDef component,
      in PublishesDef publishes)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface ComponentPublishes

  struct EventTypeLink
  {
```

```
  EventDef type;
  EventPortDef event;
};
typedef sequence<EventTypeLink> EventTypeLinkSet;

interface EventType : Reflective::RefAssociation
{
  EventTypeLinkSet all_event_type_links()
    raises (Reflective::MofError);
  boolean exists (
    in EventDef type,
    in EventPortDef event)
    raises (Reflective::MofError);
  EventDef type (in EventPortDef event)
    raises (Reflective::MofError);
  void add (
    in EventDef type,
    in EventPortDef event)
    raises (Reflective::MofError);
  void modify_type (
    in EventDef type,
    in EventPortDef event,
    in EventDef new_type)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in EventDef type,
    in EventPortDef event)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface EventType

struct APrimaryKeyHomeLink
{
  BaseIDL::ValueDef primary_key;
  HomeDef home;
};
typedef sequence<APrimaryKeyHomeLink> APrimaryKeyHomeLinkSet;

interface APrimaryKeyHome : Reflective::RefAssociation
{
  APrimaryKeyHomeLinkSet all_a_primary_key_home_links()
    raises (Reflective::MofError);
  boolean exists (
    in BaseIDL::ValueDef primary_key,
    in HomeDef home)
    raises (Reflective::MofError);
  BaseIDL::ValueDef primary_key (in HomeDef home)
    raises (Reflective::MofError);
  void add (
    in BaseIDL::ValueDef primary_key,
    in HomeDef home)
    raises (Reflective::MofError);
```

```
  void modify_primary_key (
    in BaseIDL::ValueDef primary_key,
    in HomeDef home,
    in BaseIDL::ValueDef new_primary_key)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in BaseIDL::ValueDef primary_key,
    in HomeDef home)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface APrimaryKeyHome

struct ASupportsHomeDefLink
{
  BaseIDL::InterfaceDef supports;
  HomeDef home_def;
};
typedef sequence<ASupportsHomeDefLink> ASupportsHomeDefLinkSet;

interface ASupportsHomeDef : Reflective::RefAssociation
{
  ASupportsHomeDefLinkSet all_a_supports_home_def_links()
    raises (Reflective::MofError);
  boolean exists (
    in BaseIDL::InterfaceDef supports,
    in HomeDef home_def)
    raises (Reflective::MofError);
  BaseIDL::InterfaceDefSet supports (in HomeDef home_def)
    raises (Reflective::MofError);
  void add (
    in BaseIDL::InterfaceDef supports,
    in HomeDef home_def)
    raises (Reflective::MofError);
  void modify_supports (
    in BaseIDL::InterfaceDef supports,
    in HomeDef home_def,
    in BaseIDL::InterfaceDef new_supports)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in BaseIDL::InterfaceDef supports,
    in HomeDef home_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ASupportsHomeDef

interface ComponentIDLPackageFactory
{
  ComponentIDLPackage create_component_idl_package ()
    raises (Reflective::MofError);
};

interface ComponentIDLPackage : Reflective::RefPackage
{
```

```
        readonly attribute ComponentFeatureClass component_feature_ref;
        readonly attribute ComponentDefClass component_def_ref;
        readonly attribute ProvidesDefClass provides_def_ref;
        readonly attribute HomeDefClass home_def_ref;
        readonly attribute FactoryDefClass factory_def_ref;
        readonly attribute FinderDefClass finder_def_ref;
        readonly attribute EventPortDefClass event_port_def_ref;
        readonly attribute EmitsDefClass emits_def_ref;
        readonly attribute ConsumesDefClass consumes_def_ref;
        readonly attribute UsesDefClass uses_def_ref;
        readonly attribute PublishesDefClass publishes_def_ref;
        readonly attribute EventDefClass event_def_ref;
        readonly attribute ProvidesInterface provides_interface_ref;
        readonly attribute ComponentFacet component_facet_ref;
        readonly attribute HomeFinder home_finder_ref;
        readonly attribute ComponentEmits component_emits_ref;
        readonly attribute ComponentConsumes component_consumes_ref;
        readonly attribute ComponentReceptacle component_receptacle_ref;
        readonly attribute UsesInterface uses_interface_ref;
        readonly attribute ComponentHome component_home_ref;
        readonly attribute ComponentSupports component_supports_ref;
        readonly attribute HomeFactory home_factory_ref;
        readonly attribute ComponentPublishes component_publishes_ref;
        readonly attribute EventType event_type_ref;
        readonly attribute APrimaryKeyHome a_primary_key_home_ref;
        readonly attribute ASupportsHomeDef a_supports_home_def_ref;
    };
}; // end of module ComponentIDL
```

CIF Metamodel

9

Note Ð Text in Red is from the Components 1.1 RTF interim report (ptc/2002-08-02).

Contents

This chapter contains the following sections.

Section Title	**Page**
ÒCIF PackageÓ	9-1
ÒClasses and AssociationsÓ	9-2
ÒConformance CriteriaÓ	9-6
ÒMOF DTDs and IDL for the CIF MetamodelÓ	9-7

9.1 CIF Package

In addition to the packages *BaseIDL* and *ComponentIDL*, a new package *CIF* is introduced that contains the metamodel for the Component Implementation Framework. This package obviously depends on the *ComponentIDL* package since its main purpose is to enable the modeling of implementations for components specified using the *ComponentIDL* definitions. This situation is depicted in the following diagram:

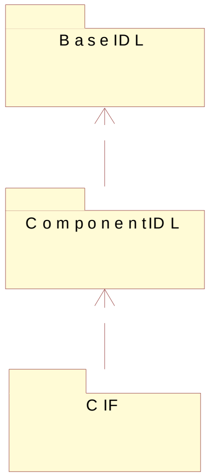


Figure 9-1 Package structure of CCM metamodels

9.2 Classes and Associations

The CIF metamodel defines additional metaclasses and associations. An overview on these is to be seen in Figure 9-2. Their meaning is explained in the following.

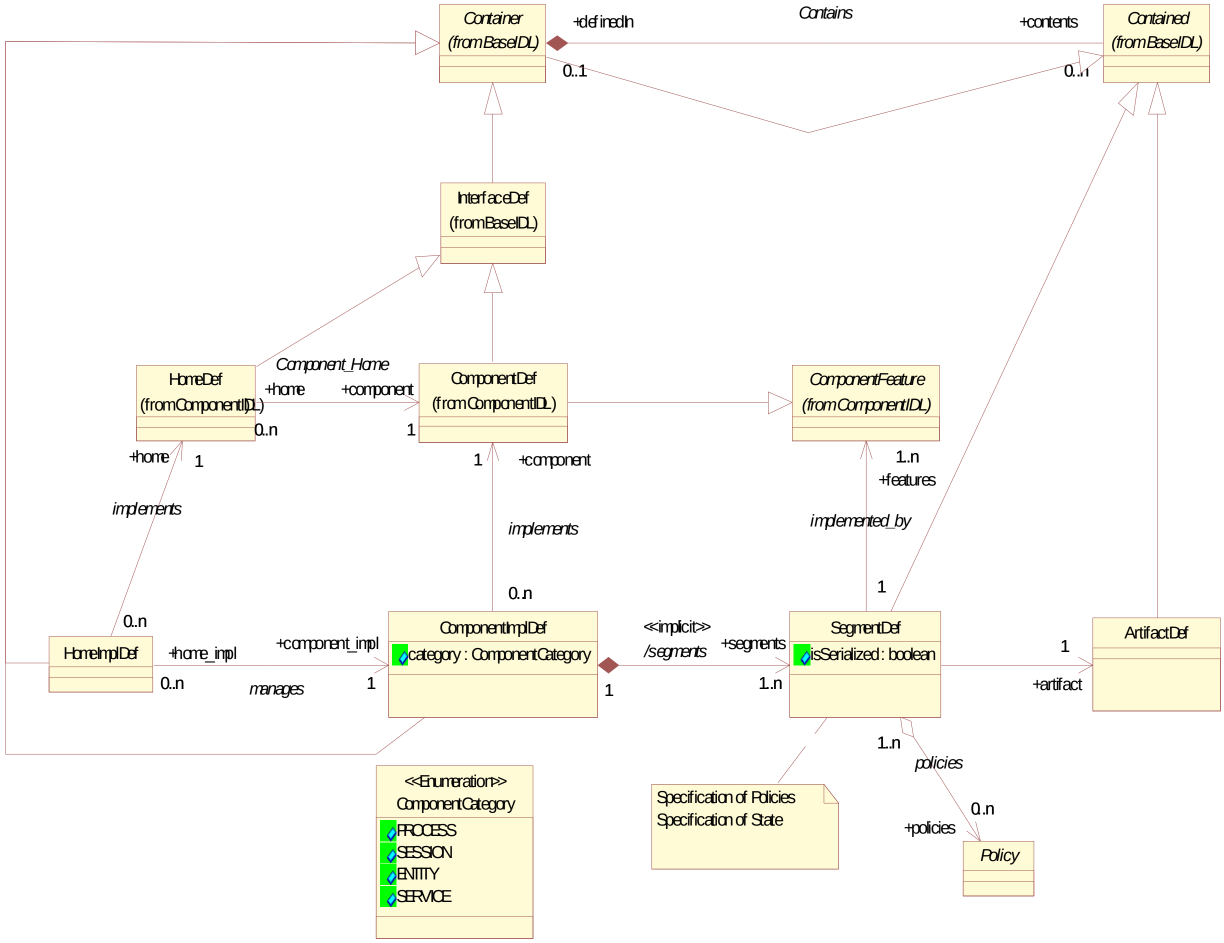


Figure 9-2 CIF metamodel (overview)

ComponentImplDef

This metaclass is used to model an implementation definition for a given component definition. It specifies an association to *ComponentDef* to allow instances to point exactly to the component the instance is going to implement. A *ComponentImplDef* always has exactly one *ComponentDef* associated while each *ComponentDef* might be implemented by different *ComponentImplDef*s.

ComponentImplDef is specified as being a *Container*, by doing so, instances are able to contain other definitions. The only definitions that are allowed to be contained by a *ComponentImplDef* are instances of *SegmentDef*.

Currently, there is no inheritance specification for instances of *ComponentImplDef*.

SegmentDef

Instances of *SegmentDef* are used to model a segmented implementation structure for a component implementation. This means that the behavior for each component feature can be provided by a separate segment of the component implementation (most likely a separate programming language class in the code generated by the CIF tools) if necessary. It is also possible to specify that a segment provides the behavior for a number of component features including the extreme that only one segment implements all component features.

Instances of *SegmentDef* are always contained in instances of *ComponentImplDef* and therefore are derived from *Contained*. *SegmentDef* has an association to *ComponentFeature* so that instances can point to all features of a component which the segment is going to implement. *SegmentDef* has in addition an association to *ArtifactDef*, which are models of programming language constructs (classes) used to actually implement the behavior for component features. There is always exactly one artifact for each segment. However, artefacts may be shared between component implementations whereas segments cannot. ThatÕs why the distinction between artefacts and segments has been modeled in the CIF.

The attribute *isSerialized* is used to indicate that the access to segment is required to be serialized or not.

ArtifactDef

ArtifactDef is used to model abstractions from programming language constructs like classes. Instances from *ArtifactDef* in a model represent the elements that provide the behavior for features of a component. Since these can be shared across component implementations the distinctions between artifacts and segments have been made in the metamodel.

ArtifactDef is a specialization of the metaclass *Contained*, which means that artifacts are identifiable and contained in other definitions. The only allowed *Container* for *ArtifactDef* is *ModuleDef*.

Policy

Segment definitions modeled as instances of the metaclass *SegmentDef* may contain a set of policies that have to be applied to realizations of the segment in the implementation code. These policies include for example activation policies for the artifact associated to a segment. The complete set of required policies is not known yet and the metamodel is designed to be flexible. *Policy* is introduced as an abstract metaclass and concrete policies are expected to be defined as specializations of that class. *SegmentDef* aggregates a set of policies.

The metamodel for component implementations is shown in Figure 9-3.

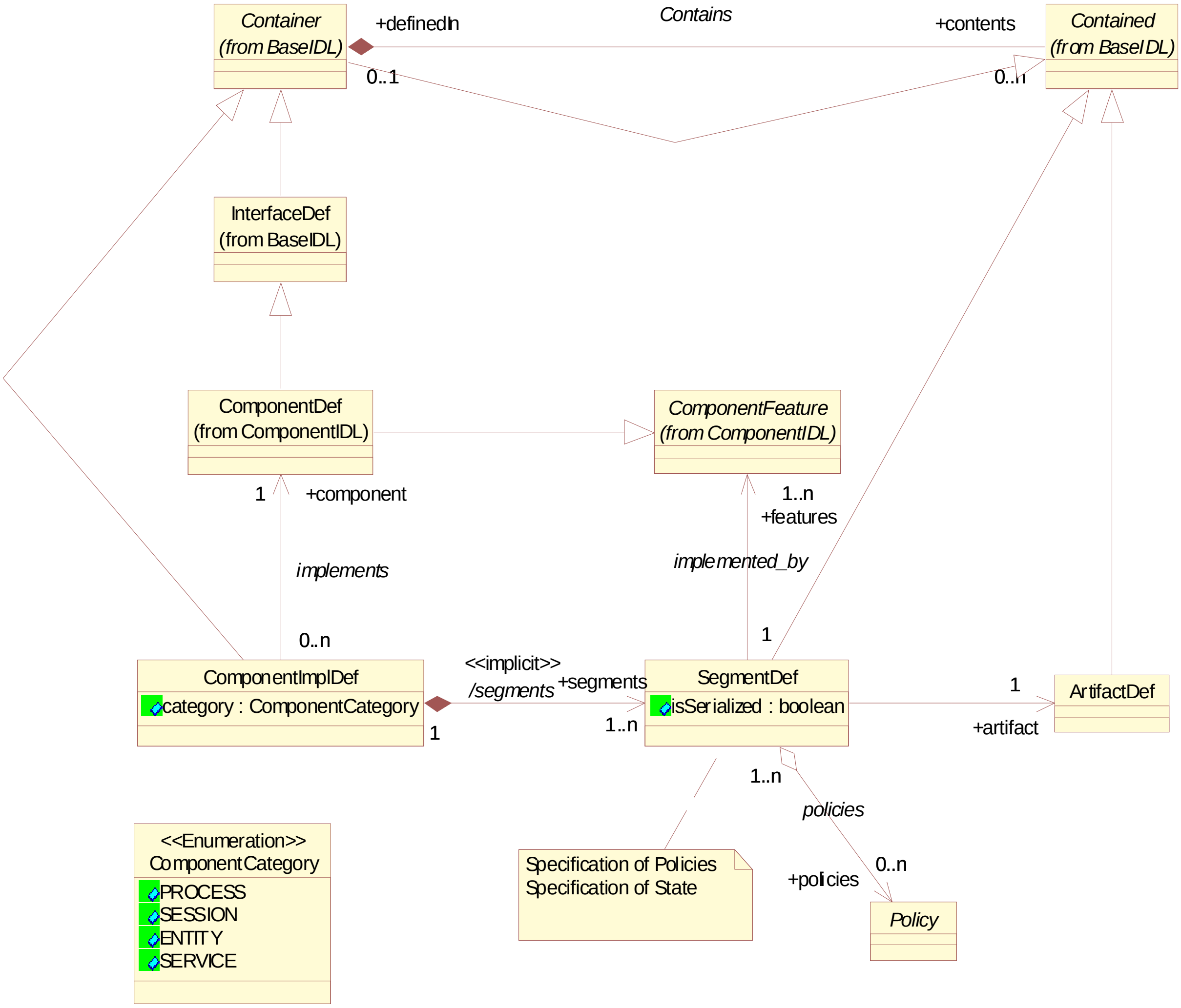


Figure 9-3 Metamodel for component implementations

HomeImplDef

Home interfaces have to be implemented and their implementation is not part of a component implementation. However, for a home implementation it has to be specified, which component implementation the home implementation manages. For that reason, the CIF metamodel contains a metaclass *HomeImplDef*. Each instance of *HomeImplDef* in a model implements exactly one instance of *HomeDef*. This relation is modeled by the association implements between both metaclasses. *HomeImplDef* inherits from the abstract metaclass *Container*. This is to allow a home implementation to be identifiable within a model and to contain other model elements. (For the time being, no such elements have been identified). Each home implementation manages exactly one component implementation, this relation is modeled by the association manages. It is required, that for each instance x of *HomeImplDef* the instance of *ComponentDef*, which is associated to the instance of *HomeDef* associated to x is the same instance as the instance of *ComponentDef* associated to the instance of *ComponentImplDef* which is associated to x.

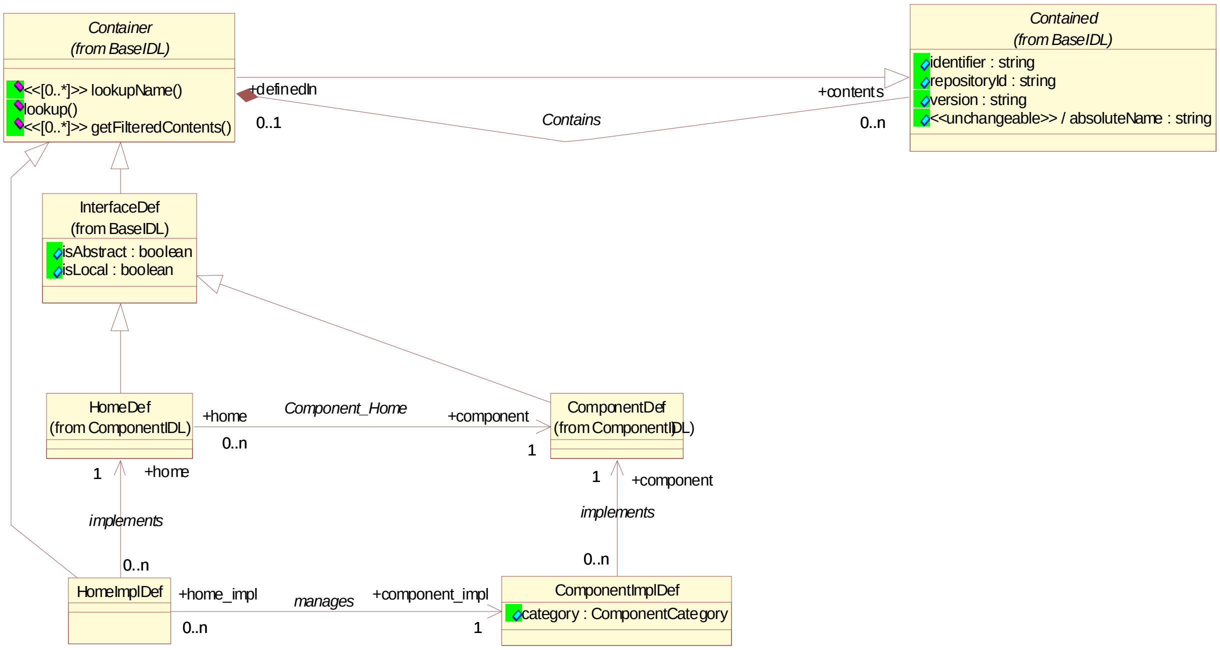


Figure 9-4 Metamodel for home implementations

9.3 Conformance Criteria

This section identifies the conformance points required for compliant implementations of the Component Implementation Framework (CIF) metamodel architecture.

9.3.1 Conformance Points

In the previous section, the MOF metamodel of the Component Implementation Framework (CIF) was defined. The following section defines the XMI format for the exchange of CIF metadata and the IDL for a MOF-compliant CIF repository. Support for the generation and consumption of the XMI metadata and for the MOF-compliant IDL is optional.

9.4 MOF DTDs and IDL for the CIF Metamodel

The XMI DTDs and IDL for the Interface Repository metamodel are presented in this section. The DTDs are generated by applying the MOF-XML mapping defined by the XMI specification to the MOF-compliant metamodel described in Section 9.1, ÒCIF Package,Ó on page 9-1. The IDL is generated by applying the MOF-IDL mapping defined in the MOF specification to the metamodels and was validated using the IDL compilers.

The IDL requires the inclusion of the reflective interfaces defined in ad/99-07-03 as part of the MOF 1.3 specification.

9.4.1 XMI DTD

See Section 8.3.1, ÒXMI DTD,Ó on page 8-27.

9.4.2 IDL for the CIF Package

```
#pragma prefix "ccm.omg.org"
#include "ComponentIDL.idl"

module CIF
{
  interface ArtifactDefClass;
  interface ArtifactDef;
  typedef sequence<ArtifactDef> ArtifactDefSet;
  interface SegmentDefClass;
  interface SegmentDef;
  typedef sequence<SegmentDef> SegmentDefSet;
  interface ComponentImplDefClass;
  interface ComponentImplDef;
  typedef sequence<ComponentImplDef> ComponentImplDefSet;
  interface PolicyClass;
  interface Policy;
  typedef sequence<Policy> PolicySet;
  interface HomeImplDefClass;
  interface HomeImplDef;
  typedef sequence<HomeImplDef> HomeImplDefSet;
  interface CIFPackage;
  enum ComponentCategory {PROCESS, SESSION, ENTITY, SERVICE};

  interface ArtifactDefClass : BaseIDL::ContainedClass
  {
    readonly attribute ArtifactDefSet all_of_type_artifact_def;
    readonly attribute ArtifactDefSet all_of_class_artifact_def;
    ArtifactDef create_artifact_def (
      in string identifier,
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface ArtifactDef : ArtifactDefClass, BaseIDL::Contained
  {
  }; // end of interface ArtifactDef

  interface SegmentDefClass : BaseIDL::ContainedClass
  {
    readonly attribute SegmentDefSet all_of_type_segment_def;
    readonly attribute SegmentDefSet all_of_class_segment_def;
    SegmentDef create_segment_def (
      in string identifier,
      in string repository_id,
      in string version,
      in boolean is_serialized)
      raises (Reflective::MofError);
```

```
  };

  interface SegmentDef : SegmentDefClass, BaseIDL::Contained
  {
    ArtifactDef artifact ()
      raises (Reflective::MofError);
    void set_artifact (in ArtifactDef new_value)
      raises (Reflective::MofError);
    ComponentIDL::ComponentFeatureSet features ()
      raises (Reflective::MofError);
    void set_features (in ComponentIDL::ComponentFeatureSet new_value)
      raises (Reflective::MofError);
    void add_features (in ComponentIDL::ComponentFeature new_element)
      raises (Reflective::MofError);
    void modify_features (
      in ComponentIDL::ComponentFeature old_element,
      in ComponentIDL::ComponentFeature new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_features (in ComponentIDL::ComponentFeature
old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    PolicySet policies ()
      raises (Reflective::MofError);
    void set_policies (in PolicySet new_value)
      raises (Reflective::MofError);
    void unset_policies ()
      raises (Reflective::MofError);
    void add_policies (in Policy new_element)
      raises (Reflective::MofError);
    void modify_policies (
      in Policy old_element,
      in Policy new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_policies (in Policy old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    boolean is_serialized ()
      raises (Reflective::MofError);
    void set_is_serialized (in boolean new_value)
      raises (Reflective::MofError);
  }; // end of interface SegmentDef

  interface ComponentImplDefClass : BaseIDL::ContainerClass
  {
    readonly attribute ComponentImplDefSet
all_of_type_component_impl_def;
    readonly attribute ComponentImplDefSet
all_of_class_component_impl_def;
    ComponentImplDef create_component_impl_def (
      in string identifier,
      in string repository_id,
      in string version,
```

```
      in ComponentCategory category)
      raises (Reflective::MofError);
  };

  interface ComponentImplDef : ComponentImplDefClass, BaseIDL::Con-
tainer
  {
    ComponentIDL::ComponentDef component ()
      raises (Reflective::MofError);
    void set_component (in ComponentIDL::ComponentDef new_value)
      raises (Reflective::MofError);
    SegmentDefSet segments ()
      raises (Reflective::MofError);
    void set_segments (in SegmentDefSet new_value)
      raises (Reflective::MofError);
    void add_segments (in SegmentDef new_element)
      raises (Reflective::MofError);
    void modify_segments (
      in SegmentDef old_element,
      in SegmentDef new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_segments (in SegmentDef old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    ComponentCategory category ()
      raises (Reflective::MofError);
    void set_category (in ComponentCategory new_value)
      raises (Reflective::MofError);
  }; // end of interface ComponentImplDef

  interface PolicyClass : Reflective::RefObject
  {
    readonly attribute PolicySet all_of_type_policy;
  };

  interface Policy : PolicyClass
  {
  }; // end of interface Policy

  interface HomeImplDefClass : BaseIDL::ContainerClass
  {
    readonly attribute HomeImplDefSet all_of_type_home_impl_def;
    readonly attribute HomeImplDefSet all_of_class_home_impl_def;
    HomeImplDef create_home_impl_def (
      in string identifier,
      in string repository_id,
      in string version)
      raises (Reflective::MofError);
  };

  interface HomeImplDef : HomeImplDefClass, BaseIDL::Container
  {
```

```
    ComponentIDL::HomeDef home ()
      raises (Reflective::MofError);
    void set_home (in ComponentIDL::HomeDef new_value)
      raises (Reflective::MofError);
    ComponentImplDef component_impl ()
      raises (Reflective::MofError);
    void set_component_impl (in ComponentImplDef new_value)
      raises (Reflective::MofError);
  }; // end of interface HomeImplDef

  struct AArtifactSegmentDefLink
  {
    ArtifactDef artifact;
    SegmentDef segment_def;
  };
  typedef sequence<AArtifactSegmentDefLink> AArtifactSegmentDefLink-
Set;

  interface AArtifactSegmentDef : Reflective::RefAssociation
  {
    AArtifactSegmentDefLinkSet all_a_artifact_segment_def_links()
      raises (Reflective::MofError);
    boolean exists (
      in ArtifactDef artifact,
      in SegmentDef segment_def)
      raises (Reflective::MofError);
    ArtifactDef artifact (in SegmentDef segment_def)
      raises (Reflective::MofError);
    void add (
      in ArtifactDef artifact,
      in SegmentDef segment_def)
      raises (Reflective::MofError);
    void modify_artifact (
      in ArtifactDef artifact,
      in SegmentDef segment_def,
      in ArtifactDef new_artifact)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ArtifactDef artifact,
      in SegmentDef segment_def)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface AArtifactSegmentDef

  struct SegmentsLink
  {
    SegmentDef segments;
    ComponentImplDef component_impl_def;
  };
  typedef sequence<SegmentsLink> SegmentsLinkSet;

  interface Segments : Reflective::RefAssociation
```

```
{
  SegmentsLinkSet all_segments_links()
    raises (Reflective::MofError);
  boolean exists (
    in SegmentDef segments,
    in ComponentImplDef component_impl_def)
    raises (Reflective::MofError);
  SegmentDefSet segments (in ComponentImplDef component_impl_def)
    raises (Reflective::MofError);
  void add (
    in SegmentDef segments,
    in ComponentImplDef component_impl_def)
    raises (Reflective::MofError);
  void modify_segments (
    in SegmentDef segments,
    in ComponentImplDef component_impl_def,
    in SegmentDef new_segments)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in SegmentDef segments,
    in ComponentImplDef component_impl_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface Segments

struct ImplementedByLink
{
  ComponentIDL::ComponentFeature features;
  SegmentDef segment_def;
};
typedef sequence<ImplementedByLink> ImplementedByLinkSet;

interface ImplementedBy : Reflective::RefAssociation
{
  ImplementedByLinkSet all_implemented_by_links()
    raises (Reflective::MofError);
  boolean exists (
    in ComponentIDL::ComponentFeature features,
    in SegmentDef segment_def)
    raises (Reflective::MofError);
  ComponentIDL::ComponentFeatureSet features (in SegmentDef
segment_def)
    raises (Reflective::MofError);
  void add (
    in ComponentIDL::ComponentFeature features,
    in SegmentDef segment_def)
    raises (Reflective::MofError);
  void modify_features (
    in ComponentIDL::ComponentFeature features,
    in SegmentDef segment_def,
    in ComponentIDL::ComponentFeature new_features)
    raises (Reflective::NotFound, Reflective::MofError);
```

```
  void remove (
    in ComponentIDL::ComponentFeature features,
    in SegmentDef segment_def)
  raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface ImplementedBy

struct PoliciesLink
{
  Policy policies;
  SegmentDef segment_def;
};
typedef sequence<PoliciesLink> PoliciesLinkSet;

interface Policies : Reflective::RefAssociation
{
  PoliciesLinkSet all_policies_links()
    raises (Reflective::MofError);
  boolean exists (
    in Policy policies,
    in SegmentDef segment_def)
    raises (Reflective::MofError);
  PolicySet policies (in SegmentDef segment_def)
    raises (Reflective::MofError);
  void add (
    in Policy policies,
    in SegmentDef segment_def)
    raises (Reflective::MofError);
  void modify_policies (
    in Policy policies,
    in SegmentDef segment_def,
    in Policy new_policies)
    raises (Reflective::NotFound, Reflective::MofError);
  void remove (
    in Policy policies,
    in SegmentDef segment_def)
    raises (Reflective::NotFound, Reflective::MofError);
}; // end of interface Policies

struct ManagesLink
{
  ComponentImplDef component_impl;
  HomeImplDef home_impl;
};
typedef sequence<ManagesLink> ManagesLinkSet;

interface Manages : Reflective::RefAssociation
{
  ManagesLinkSet all_manages_links()
    raises (Reflective::MofError);
  boolean exists (
    in ComponentImplDef component_impl,
```

```
      in HomeImplDef home_impl)
      raises (Reflective::MofError);
    ComponentImplDef component_impl (in HomeImplDef home_impl)
      raises (Reflective::MofError);
    void add (
      in ComponentImplDef component_impl,
      in HomeImplDef home_impl)
      raises (Reflective::MofError);
    void modify_component_impl (
      in ComponentImplDef component_impl,
      in HomeImplDef home_impl,
      in ComponentImplDef new_component_impl)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (
      in ComponentImplDef component_impl,
      in HomeImplDef home_impl)
      raises (Reflective::NotFound, Reflective::MofError);
  }; // end of interface Manages

  interface CIFPackageFactory
  {
    CIFPackage create_cif_package ()
      raises (Reflective::MofError);
  };

  interface CIFPackage : Reflective::RefPackage
  {
    readonly attribute ArtifactDefClass artifact_def_ref;
    readonly attribute SegmentDefClass segment_def_ref;
    readonly attribute ComponentImplDefClass component_impl_def_ref;
    readonly attribute PolicyClass policy_ref;
    readonly attribute HomeImplDefClass home_impl_def_ref;
    readonly attribute AArtifactSegmentDef a_artifact_segment_def_ref;
    readonly attribute Segments segments_ref;
    readonly attribute ImplementedBy implemented_by_ref;
    readonly attribute Policies policies_ref;
    readonly attribute Manages manages_ref;
  };
}; // end of module CIF
```

Index

A
Abstract spec 2-16
Abstract Storage home binding 3-3
accessmode Element 6-22
application object 1-ix
Application operation 4-15
ArtifactDef 9-3
Assembly interface 6-73
AssemblyFactory interface 6-72
Attributes 1-3
author Element 6-5

B
BaseIDL Package 8-2, 8-59
Basic components 1-6

Note Ð Issue 5499

C
Callback interfaces 5-26
~~catalog Element 6-22~~
~~Catalog usage declaration 2-7~~
CCM2Context interface 4-33
CCMContext interface 4-22
CCMHome interface 1-41
CCMObject interface 1-52
choice Element 6-62
choices Element 6-62
CIDL Compiler responsibilities 6-18
CIDL specification 2-5
CIF Package 9-1
CIF package IDL 9-7
Classes and Associations 9-2
Client Programming Model 4-45
code Element 6-5
codebase Element 6-6, 6-44
Common Facilities ix
company Element 6-6
Comparing CCM and EJB 5-23
compiler Element 6-6
compliance x
Component 1-2
  activation 4-8
  Assembly Descriptor 6-40
  Assembly File 6-40
  Assembly Packaging 6-39
  body 1-7
  categories 4-5
  configuration 1-44
  containers 4-5
  deployment 6-66
  Entry Points (Component Home Factories) 6-78
  executor 3-3
  factory 4-8
  home 3-3
  homes 1-5
  Identifiers 4-38
  identity 1-4, 1-12
  implementation 4-15
  Implementation Definition Language (CIDL) 2-1
  Implementation Framework (CIF) 3-1
  inheritance 1-51
  Interfaces 4-21
  interfaces 5-25
  levels 1-2, 4-15
  Object References 3-36
  packaging 6-2
  type 1-2
componentassembly Root Element 6-43
componentassembly.dtd 7-13
Component-aware clients 4-46
componentfeatures Element 6-23
componentfile Element 6-44
componentfileref Element 6-44
componentfiles Element 6-45
ComponentId interface 4-42
ComponentIDL Package 8-14
ComponentImplDef 9-2
componentimplref Element 6-45
ComponentInstallation interface 6-71
componentinstantiation Element 6-45
componentinstantiationref Element 6-46
componentkind Element 6-23
componentproperties Element 6-24, 6-46
componentrepid Element 6-24
ComponentServer interface 6-75
componentsupportedinterface Element 6-46
Component-unaware clients 4-50
Composition 3-3
Composition definition 2-5
Composition structure 3-4
Configuration phase 1-46
Configuration with attributes 1-47
configurationcomplete Element 6-24
Configurator interface 1-47
Conformance criteria 8-26, 9-5
Conformance requirements 1-54
Connect operations 1-16
connectevent Element 6-47
connecthomes Element 6-47
connectinterface Element 6-47
Connection 1-27
connections Element 6-48
Constraints on basic components 8-24
consumes Element 6-24
consumesidentifier Element 6-48
consumesport Element 6-49
Container 1-23
Container API type 4-4
Container interface 6-76
Container-managed Persistence 4-14
containermanagedpersistence Element 6-25
Containers and Contained Elements 8-16
Containment 8-6
Containment constraints 8-7
Context interfaces 5-28
Cookie values 1-18
CORBA
  contributors xi
  documentation set ix
  Component descriptor 6-16

Index

  - component views for EJBs 5-4
  - usage model 4-4, 4-6
- corbacomponent Element 6-25
- corbacomponent Root Element 6-21
- corbacomponent.dtd 7-5
- corbaversion Element 6-26
- core, compliance x

D

- defaultvalue Element 6-62
- Delegation of feature state 3-34
- Delegation specification 3-4
- dependency Element 6-6
- Deployment 6-67
- Deployment information 6-17
- description Element 6-7, 6-26, 6-49, 6-63
- descriptor Element 6-7
- destination Element 6-49
- destinationhome Element 6-49
- Disconnect operations 1-17

E

- EJB to IDL mapping 5-6
- EJB views for CORBA components 5-14
- emits Element 6-26
- emitsidentifier Element 6-50
- emitsport Element 6-50
- EnterpriseComponent interface 4-27
- Entity component 4-19
- entity Element 6-26
- Entity2Context interface 4-44
- EntityComponent interface 4-31
- EntityContext interface 4-30
- Entry points 3-47
- entrypoint Element 6-7
- Equivalent IDL 1-6
- Equivalent interface 1-3, 1-6
- Event
  - emitter operations 1-27
  - interface 4-36
  - publisher operations 1-25
  - sink operations 1-28
  - sinks 1-2, 1-27
  - sources 1-2
  - types 1-21
- EventBase 1-22
- EventConsumer interface 1-22
- eventpolicy Element 6-26
- Events 4-12, 4-50
- Events interface 1-29
- Exceptions 8-11
- executableplacement Element 6-50
- Executor 3-2
- Executor definition 2-10, 3-26
- Executor delegation specification 2-15
- existinginterface Element 6-51
- extendedpoapolicy Element 6-27
- extension Element 6-8, 6-27, 6-51
- External API types 4-4

F

- Facet 1-8
- Facet declaration 2-12
- Facets 1-2, 1-4
- Factories 3-47
- Factory design pattern 4-47
- Factory operation 1-37
- Factory-based configuration 1-48
- Feature delegation specification 2-12
- fileinarchive Element 6-8, 6-27, 6-51
- findby Element 6-51
- Finder design pattern 4-48
- Finder operation 1-37
- Finders 3-47

G

- Grammar 2-3

H

- Heterodox operations 1-40
- Home 3-45
  - definitions with no primary key 1-33
  - definitions with primary keys 1-34
  - executor 3-3
  - executor definition 2-8
  - factories 3-47
  - implementation declaration 2-9
  - inheritance 1-38
  - interface 4-23
  - operations 1-39
  - persistence declaration 2-10
- HomeConfiguration interface 1-49
- homefeatures Element 6-27
- HomeFinder 1-42
- homefinder Element 6-52
- HomeImplDef 9-4
- homeinterface Element 6-52
- homeplacement Element 6-52
- homeplacementref Element 6-53
- homeproperties Element 6-28, 6-53
- HomeRegistration interface 4-34
- homerepid Element 6-28
- Homes 1-32
- hostcollocation Element 6-54
- humanlanguage Element 6-9

I

- idl Element 6-9
- implementation Element 6-9
- Implementations 3-2
- implref Element 6-10
- impltype Element 6-54
- inheritscomponent Element 6-28
- inheritshome Element 6-28
- inheritsinterface Element 6-29
- ins Element 6-29
- interface Element 6-29
- Interface repository 1-2
- Interfaces common to both Container API types 4-22
- Interfaces supported by the Entity Container API type 4-30, 4-38
- Interfaces supported by the Session Container API type 4-27, 4-36

interop Element 6-30
interoperability, compliance x
interworking
  compliance x
invocation Element 6-54

J
Java Language to IDL mapping 5-5

K
KeylessCCMHome interface 1-42
Keywords 2-3

L
Language Mapping 3-38
license Element 6-10
Life cycle 2-6
link Element 6-10, 6-30, 6-55
localfile Element 6-11

M
Managed storage 3-11
Mapping EJB to Component IDL definitions 5-5
Mapping rules 3-40
Metadata interfaces 5-30

N
name Element 6-11
Naming 8-13
namingservice Element 6-55
Navigation 1-9

O
Object Management Group vii
  address of ix
Object Request Broker viii
Object Services ix
objref Element 6-30
ompliance with the interoperability of integration views 5-22
operation Element 6-30
Operational phase 1-46
operationpolicies Element 6-31
Operations 8-13
Orthodox operations 1-40
os Element 6-11

P
param Element 6-31
params Element 6-31
partitioning Element 6-55
Persistence 4-13
Persistent objects 4-7
Persistent State Definition Language 3-2
PersistentIdValue 4-41
pkgtype Element 6-12
poapolicies Element 6-32
Policies 4-12, 4-13
Policy 9-3
Ports 1-2, 3-43
ports Element 6-32
Primary key 1-36
Primary key binding 3-14
PrimaryKeyBase 1-36
Process component 4-18
process Element 6-32
processcollocation Element 6-56
processor Element 6-12
programminglanguageElement 6-13
properties Element 6-63
properties Root Element 6-62
properties.dtd 7-11
Property File Descriptor 6-61
propertyfile Element 6-13
provides Element 6-33
providesidentifier Element 6-56
providesport Element 6-56
Proxy
  home 3-4
  home declaration 2-16
  Homes 3-35
proxyhome Element 6-58
ProxyHomeRegistration interface 4-35
pssimplementation Element 6-33
psstransaction Element 6-33
psstransactionisolationlevel Element 6-34
Publishers and Emitters 1-23
publishes Element 6-34
publishesidentifier Element 6-57
publishesport Element 6-57

R
range Element 6-63
Receptacles 1-2, 1-15
Receptacles Interface 1-18
reference model 1-viii
References 1-12
registercomponent Element 6-57
registerwithhomefinder Element 6-58
registerwithnaming Element 6-58
registerwithtrader Element 6-58
Relationship between Home Executor and Abstract Storage Home 3-13
repository Element 6-13, 6-34
requiredrights Element 6-35
right Element 6-35
runtime Element 6-14

S
Security 4-11, 4-49
security Element 6-35
Segment definition 2-10
segment Element 6-36
Segment persistence declaration 2-11
SegmentDef 9-3
SegmentDescr 4-42
Segmented executors 3-27
segmentmember Element 6-36
Self-managed Persistence 4-14
sequence Element 6-64
servant Element 6-36
Servant lifetime management 4-8
Servant to ObjectId mapping 4-7
Server programming environment 4-5

Index

Server Programming interfaces 4-21
Server programming interfaces - Extended Components 4-33
ServerActivator interface 6-74
Service component 4-16
service Element 6-37
Session component 4-17
session Element 6-37
Session2Context interface 4-37
SessionComponent interface 4-28
SessionContext interface 4-27
SessionSynchronization interface 4-29
simple Element 6-63
simple-link-attributes Entity 6-14, 6-37
softpkg Descriptor 6-3
softpkg Element 6-15
softpkg Root Element 6-4
softpkg.dtd 7-1
softpkgref Element 6-15
StandardConfigurator interface 1-48
StateIdFactory 4-41
StateIdValue type 4-40
Storage home binding 2-9
Storage home delegation specification 2-13
storagehome Element 6-37
stringifiedobjectref Element 6-59
struct Element 6-65
Subscription 1-27
Supported interfaces 1-13
supportsinterface Element 6-37

T
Threading 4-7
threading Element 6-38
title Element 6-15
Tools 1-55
Trader elements 6-59
transaction Element 6-38
Transaction interfaces 5-29
Transactions 4-9, 4-48
Transient objects 4-7
Translating CORBA Component requests into EJB requests 5-9
Translating EJB requests into CORBA component requests 5-17
Typedef and Type derivations 8-10
Typing 8-5

U
unclassified Element 6-38
usage Element 6-15
usagename Element 6-60
UserTransaction interface 4-23
uses Element 6-39
usesidentifier Element 6-60
usesport Element 6-60

V
value Element 6-65
Value types 8-12
ValueDef Constraints EventsDef 8-21
valuetype Element 6-65
valuetypefactory Element 6-15

W
webpage Element 6-16

X
XMI DTD 8-27, 9-6
XML DTDs 7-1
XML Elements 6-4, 6-20, 6-43, 6-62