

BS ISO/IEC 19500-3:2012



BSI Standards Publication

Information technology — Object Management Group — Common Object Request Broker Architecture (CORBA)

Part 3: Components

bsi.

...making excellence a habit.
AVA Documents™
Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 19500-3:2012.

The UK participation in its preparation was entrusted to Technical Committee IST/15, Software and systems engineering.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2012. Published by BSI Standards Limited 2012

ISBN 978 0 580 65982 9

ICS 35.080

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 November 2012.

Amendments issued since publication

Date	Text affected
------	---------------

**Information technology — Object
Management Group — Common Object
Request Broker Architecture (CORBA) —
Part 3:
Components**

*Technologies de l'information — OMG (Object Management Group) —
CORBA (Common Object Request Broker Architecture) —*

Partie 3: Composants





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2012

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Table of Contents

Foreword	xi
Introduction	xiii
1 Scope	1
2 Conformance and Compliance	1
3 References	3
3.1 Normative References	3
3.2 Non-normative References	4
4 Terms and definitions	4
4.1 Terms Defined in this International Standard	4
4.2 Keywords for Requirement statements	7
5 Symbols (and abbreviated terms)	7
6 Component Model	9
6.1 Component Model	9
6.1.1 Component Levels	9
6.1.2 Ports	9
6.1.3 Components and Facets	10
6.1.4 Component Identity	11
6.1.5 Component Homes	11
6.2 Component Definition	11
6.3 Component Declaration	11
6.3.1 Basic Components	11
6.3.2 Equivalent IDL	12
6.3.3 Component Body	13
6.4 Facets and Navigation	13
6.4.1 Equivalent IDL	13
6.4.2 Semantics of Facet References	14

6.4.3 Navigation	14
6.4.4 Provided References and Component Identity	17
6.4.5 Supported interfaces	18
6.5 Receptacles	20
6.5.1 Equivalent IDL	20
6.5.2 Behavior	21
6.5.3 Receptacles Interface	22
6.6 Events	25
6.6.1 Event types	25
6.6.2 EventConsumer Interface	26
6.6.3 Event Service Provided by Container	27
6.6.4 Event Sources—Publishers and Emitters	27
6.6.5 Publisher	28
6.6.6 Emitters	29
6.6.7 Event Sinks	30
6.6.8 Events interface	30
6.7 Homes	34
6.7.1 Equivalent Interfaces	34
6.7.2 Primary Key Declarations	36
6.7.3 Explicit Operations in Home Definitions	37
6.7.4 Home inheritance	38
6.7.5 Semantics of Home Operations	39
6.7.6 CCMHome Interface	41
6.7.7 KeylessCCMHome Interface	42
6.8 Home Finders	42
6.9 Component Configuration	44
6.9.1 Exclusive Configuration and Operational Life Cycle Phases	45
6.10 Configuration with Attributes	46
6.10.1 Attribute Configurators	46
6.10.2 Factory-based Configuration	47
6.11 Component Inheritance	49
6.11.1 CCMObject Interface	50
6.12 Conformance Requirements	51
6.12.1 A Note on Tools	53
6.12.2 Changes to Object Services	53
7 OMG CIDL Syntax and Semantics	55
7.1 General	55

7.2 Lexical Conventions	55
7.2.1 Keywords	56
7.3 OMG CIDL Grammar	56
7.4 OMG CIDL Specification	58
7.5 Composition Definition	58
7.5.1 Life Cycle Category and Constraints	59
7.6 Home Executor Definition	59
7.7 Home Implementation Declaration	60
7.8 Storage Home Binding	61
7.9 Home Persistence Declaration	61
7.10 Executor Definition	61
7.11 Segment Definition	62
7.12 Segment Persistence Declaration	62
7.13 Facet Declaration	63
7.14 Feature Delegation Specification	63
7.15 Abstract Storage Home Delegation Specification	64
7.16 Executor Delegation Specification	65
7.17 Abstract Spec Declaration	66
7.18 Proxy Home Declaration	66
8 CCM Implementation Framework	67
8.1 Introduction	67
8.2 Component Implementation Framework (CIF) Architecture	67
8.2.1 Component Implementation Definition Language (CIDL)	67
8.2.2 Component persistence and behavior	67
8.2.3 Implementing a CORBA Component	67
8.2.4 Behavioral elements: Executors	68
8.2.5 Unit of implementation : Composition	68
8.2.6 Composition structure	69
8.2.7 Compositions with Managed Storage	75
8.2.8 Relationship between Home Executor and Abstract Storage Home	77
8.2.9 Executor Definition	89
8.2.10 Proxy Homes	96
8.2.11 Component Object References	97

8.3 Language Mapping	99
8.3.1 Overview	99
8.3.2 Common Interfaces	100
8.3.3 Mapping Rules	101
9 The Container Programming Model	109
9.1 General	109
9.2 Introduction	109
9.2.1 External API Types	110
9.2.2 Container API Type	111
9.2.3 CORBA Usage Model	111
9.2.4 Component Categories	111
9.3 The Server Programming Environment	112
9.3.1 Component Containers	112
9.3.2 CORBA Usage Model	113
9.3.3 Component Factories	114
9.3.4 Component Activation	114
9.3.5 Servant Lifetime Management	114
9.3.6 Transactions	115
9.3.7 Security	117
9.3.8 Events	117
9.3.9 Persistence	118
9.3.10 Application Operation Invocation	119
9.3.11 Component Implementations	120
9.3.12 Component Levels	120
9.3.13 Component Categories	120
9.4 Server Programming Interfaces - Basic Components	124
9.4.1 Component Interfaces	124
9.4.2 Interfaces Common to both Container API Types	125
9.4.3 Interfaces Supported by the Session Container API Type	130
9.4.4 Interfaces Supported by the Entity Container API Type	132
9.5 Server Programming Interfaces - Extended Components	134
9.5.1 Interfaces Common to both Container API Types	134
9.5.2 Interfaces Supported by the Session Container API Type	136
9.5.3 Interfaces Supported by the Entity Container API Type	138
9.6 The Client Programming Model	144
9.6.1 Component-aware Clients	144
9.6.2 Component-unaware Clients	148

10	Integrating with Enterprise JavaBeans	151
10.1	Introduction	151
10.2	Enterprise JavaBeans Compatibility Objectives and Requirements	152
10.3	CORBA Component Views for EJBs	153
10.3.1	Mapping of EJB to Component IDL definitions	153
10.3.2	Translation of CORBA Component requests into EJB requests	157
10.3.3	Interoperability of the View	158
10.3.4	CORBA Component view Example	160
10.4	EJB views for CORBA Components	162
10.4.1	Mapping of Component IDL to Enterprise JavaBeans specifications	162
10.4.2	Translation of EJB requests into CORBA Component Requests	164
10.4.3	Interoperability of the View	166
10.4.4	Example	168
10.5	Compliance with the Interoperability of Integration Views	169
10.6	Comparing CCM and EJB	169
10.6.1	The Home Interfaces	170
10.6.2	The Component Interfaces	171
10.6.3	The Callback Interfaces	173
10.6.4	The Context Interfaces	174
10.6.5	The Transaction Interfaces	175
10.6.6	The Metadata Interfaces	176
11	Interface Repository Metamodel	177
11.1	Introduction	177
11.1.1	BaseIDL Package	177
11.1.2	ComponentIDL Package	188
11.2	Conformance Criteria	196
11.2.1	Conformance Points	197
11.3	MOF DTDs and IDL for the Interface Repository Metamodel	197
11.3.1	XMI DTD	197
11.3.2	IDL for the BaseIDL Package	222
11.3.3	IDL for the ComponentIDL Package	244
12	CIF Metamodel	263
12.1	CIF Package	263
12.2	Classes and Associations	263

12.2.1 ComponentImplDef	264
12.2.2 SegmentDef	265
12.2.3 ArtifactDef	265
12.2.4 Policy	265
12.2.5 HomeImplDef	266
12.3 Conformance Criteria	267
12.3.1 Conformance Points	267
12.4 MOF DTDs and IDL for the CIF Metamodel	267
12.4.1 XMI DTD	268
12.4.2 IDL for the CIF Package	268
13 Lightweight CCM Profile	275
13.1 Summary	275
13.2 Changes associated with excluding support for persistence	276
13.3 Changes associated with excluding support for introspection, navigation and type-specific operations redundant with generic operations	278
13.4 Changes associated with excluding support for segmentation ...	279
13.5 Changes associated with excluding support for transactions	280
13.6 Changes associated with excluding support for security	280
13.7 Changes associated with excluding support for configurators ...	281
13.8 Changes associated with excluding support for proxy homes	281
13.9 Changes associated with excluding support for home finders	281
13.10 Changes adding additional restrictions to the extended model not represented by exclusions above	282
14 Deployment PSM for CCM	283
14.1 Overview	283
14.2 Definition of Meta-Concepts	284
14.2.1 Component	284
14.2.2 ImplementationArtifact	285
14.2.3 PackageI	285
14.3 PIM to PSM for CCM Transformation	285
14.3.1 ComponentInterfaceDescription	285

14.3.2 PlanSubcomponentPortEndpoint	286
14.3.3 Application	286
14.3.4 RepositoryManager	287
14.3.5 SatisfierProperty	287
14.4 PSM for CCM to PSM for CCM for IDL Transformation	287
14.4.1 Generic Transformation Rules	287
14.4.2 Special Transformation Rules	289
14.4.3 Mapping to IDL	290
14.5 PSM for CCM to PSM for CCM for XML Transformation	290
14.5.1 Generic Transformation Rules	290
14.5.2 Special Transformation Rules	291
14.5.3 Transformation Exceptions and Extensions	295
14.5.4 Interpretation of Relative References	296
14.5.5 Mapping to XML	297
14.6 Miscellaneous	297
14.6.1 Entry Points	297
14.6.2 Homes	298
14.6.3 Valuetype Factories	298
14.6.4 Discovery and Initialization	298
14.6.5 Location	299
14.6.6 Segmentation	299
14.7 Migration Issues	300
14.7.1 Component Implementations	300
14.7.2 Component and Assembly Packages and Metadata	300
14.7.3 Component Deployment Systems	300
14.8 Metadata Vocabulary	301
14.8.1 Implementation Selection Requirements	301
14.8.2 Monolithic Implementation Resource Requirements	301
15 Deployment IDL for CCM	303
16 XML Schema for CCM	317
Annex A - Legal Information	337

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19500-3 was prepared by Technical Committee ISO/IEC JTC1, Information technology, in collaboration with the Object Management Group (OMG), following the submission and processing as a Publicly Available Specification (PAS) of the OMG Common Object Request Broker Architecture (CORBA) specification Part 3 Version 3.1 CORBA Components.

ISO/IEC 19500-3 is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, Information Technology - Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language
- ISO/IEC 19500-2, Information Technology - Open Distributed Processing - CORBA Specification Part 1: CORBA Interfaces
- ISO/IEC 19500-3, Information Technology - Open Distributed Processing - CORBA Specification Part 2: CORBA Interoperability

ISO/IEC 19500 consists of the following parts, under the general title *Information technology - Open distributed processing - CORBA specification*:

- Part 1: CORBA Interfaces
- Part 2: CORBA Interoperability
- Part 3: CORBA Components

It is the common core of the CORBA specification. Optional parts of CORBA, such as mappings to particular programming languages, Real-time CORBA extensions, and the minimum CORBA profile for embedded systems are documented in the other specifications that together comprise the complete CORBA specification. Please visit the CORBA download page at http://www.omg.org/technology/documents/corba_spec_catalog.htm to find the complete CORBA specification set.

Apart from this Foreword, the text of this International Standard is identical with that for the OMG specification for CORBA, v3.1.1, Part 3.

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 3 of the RM-ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This part of ISO/IEC 19500 (CORBA Components) is a standard for the technology specification of an ODP system. It defines a technology to provide the infrastructure required to support functional distribution of an ODP system, specifying functions required to manage physical distribution, communications, processing and storage, and the roles of different technology objects in supporting those functions.

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 *CORBA services: 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 *CORBA facilities: Common Facilities Architecture*.
- **Application Objects**, which are products of a single vendor or 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.

The architecture and specifications described in this standard are aimed at software designers and developers who want to produce applications that comply with OMG specifications for the Object Request Broker (ORB), or this standard (ISO/IEC 19500). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. 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.

This Part of this International Standard includes a non-normative annex.

Information technology - Object Management Group Common Object Request Broker Architecture (CORBA), Components

1 Scope

This part of ISO/IEC 19500 defines:

- The syntax and semantics of a component model (see Clause 6, 'Component Model'), based on CORBA IDL, and a corresponding meta-model (see Clause 11, 'Interface Repository Metamodel').
- A language to describe the structure and state of component implementations (see Clause 7, 'OMG CIDL Syntax and Semantics'), and a corresponding meta-model (see Clause 12, 'CIF Metamodel').
- A programming model for constructing component implementations (see Clause 8, 'CCM Implementation Framework').
- A runtime environment for component implementations (see Clause 9, 'The Container Programming Model').
- Interaction between components and Enterprise Java Beans (see Clause 10, 'Integrating with Enterprise JavaBeans').
- Meta-data for describing component-based applications, and interfaces for their deployment (see Clause 14, 'Deployment PSM for CCM').
- A lightweight subset of the component model, programming model and runtime environment (see Clause 13, 'Lightweight CCM Profile').

2 Conformance and Compliance

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 "Changes to Object Services" on page 53.
2. A CORBA Component vendor shall provide a conforming implementation of the Basic Level of CORBA Components. A Lightweight CORBA Component vendor shall provide a conforming implementation of the Lightweight CCM Profile as specified in item 8 below.
3. A CORBA Component vendor may provide a conforming implementation of the Extended Level of CORBA Components.
4. 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 ("Segment Definition" on page 62) 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.

Such implementations shall work on a CORBA ORB as defined in #1 above.

5. 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 “Translation of CORBA Component requests into EJB requests” on page 157.

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

6. 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 “Deployment PSM for CCM” on page 283.

Such implementations shall work on a CORBA ORB as defined in #1 above.

7. The Lightweight CCM profile is a conformance point based on the extended model as defined above. “Lightweight CCM Profile” on page 275 defines the specific parts of this CCM specification that are impacted and the normative specific subsetting of CCM. In summary, the following general capabilities (and associated machinery) are excluded from the extended model to define this conformance point:

- Persistence (only session and service components are supported)
- Introspection
- Navigation
- Redundancies, preferring generic over specific
- Segmentation (not allowed for session or service components)
- Transactions
- Security
- Configurators
- Proxy homes
- Home finders
- CIDL
- POA related mandates

8. A CORBA Component vendor may optionally support EJB clients interacting with CORBA Components, by implementing the IDL to EJB mapping as defined in “Translation of EJB requests into CORBA Component Requests” on page 164.

3 References

3.1 Normative References

- [CORBA] Object Management Group, "Common Object Request Broker Architecture," version 3.0.3, OMG document number formal/04-03-01. Available from <http://www.omg.org/cgi-bin/doc?formal/04-03-01>
- [D+C] Object Management Group, "*Deployment and Configuration of Component-based Distributed Applications Specification*," version 1.1. OMG document number ptc/05-01-07. Available from <http://www.omg.org/cgi-bin/doc?ptc/05-01-07>
- [EJB] Java Community Process, "Enterprise JavaBeans Specification - Version 1.1" (with errata), June 8, 2000. Available from <http://jcp.org/aboutJava/communityprocess/maintenance/jsr905>
- [HTTP] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, RFC 2616: "*Hypertext Transfer Protocol -- HTTP/1.1*." June 1999. Available from <http://www.ietf.org/rfc/rfc2616.txt>
- [INS] Object Management Group, "*Naming Service Specification*," version 1.1. February 2001. OMG document number formal/01-02-65. Available from <http://www.omg.org/cgi-bin/doc?formal/01-02-65>
- [JIDL] Object Management Group, "Java™ to IDL Language Mapping Specification," version 1.3. September 2003. OMG document number formal/03-09-04. Available from <http://www.omg.org/cgi-bin/doc?formal/03-09-04>
- [MDA] Object Management Group, "MDA Guide - Version 1.0.1" Available from <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
- [MOF1] ISO/IEC 19503:2005 Information technology -- XML Metadata Interchange (XMI)
- [PSS] Object Management Group, "*Persistent State Service*," version 2.0. September 2002. OMG document number formal/02-09-06. Available from <http://www.omg.org/cgi-bin/doc?formal/02-09-06>
- [RFC2119] IETF RFC2119, "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. Available from <http://ietf.org/rfc/rfc2119>
- [SSS] Object Management Group, "*Security Service Specification*," version 1.8. March 2002. OMG document number formal/02-03-11. Available from <http://www.omg.org/cgi-bin/doc?formal/02-03-11>
- [TSS] Object Management Group, "*Transaction Service Specification*," version 1.4. September 2003. OMG document number formal/03-09-02. Available from <http://www.omg.org/cgi-bin/doc?formal/03-09-02>

[UML1]	/IEC 19501:2005 Information technology - Unified Modeling Language (UML)
[UPC]	Object Management Group, “UML™ Profile for CORBA™ Specification,” version 1.0. Adopted specification. April 2002. OMG document number formal/02-04-01. Available from http://www.omg.org/cgi-bin/doc?formal/02-04-01
[URI]	T. Berners-Lee, R. Fielding, L. Masinter, RFC 2396: “Uniform Resource Identifiers (URI): Generic Syntax.” August 1998, available from http://www.ietf.org/rfc/rfc2396.txt
[XMI]	ISO/IEC 19503:2005 Information technology -- XML Metadata Interchange (XMI)
[XML]	World Wide Web Consortium (W3C), “Extensible Markup Language (XML),” version 1.0 (second edition). W3C Recommendation, October 6, 2000. Available from http://www.w3.org/TR/REC-xml
[XSD]	World Wide Web Consortium (W3C), “XML Schema Part 1: Structures.” W3C Recommendation, May 2, 2001. Available from http://www.w3.org/TR/xmlschema-1/ World Wide Web Consortium (W3C), “XML Schema Part 2: Datatypes.” W3C Recommendation, May 2, 2001. Available from http://www.w3.org/2001/xmlschema-2/

3.2 Non-normative References

[LCS]	Object Management Group, “Life Cycle Service Specification,” version 1.2. September 2002. OMG document number formal/02-09-01. Available from http://www.omg.org/cgi-bin/doc?formal/02-09-01
[NSS]	Object Management Group, “Notification Service Specification,” version 1.1. October 2004. OMG document number formal/04-10-11. Available from http://www.omg.org/cgi-bin/doc?formal/04-10-11

4 Terms and definitions

4.1 Terms Defined in this International Standard

Basic Component

A basic component is not allowed to inherit from other components, offer facets, receptacles, event sources or sinks. A basic component may only offer attributes.

Component

A specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository.

Component Home

A 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.

Component-aware Client

A client that is defined using the IDL extensions in the component model.

Composition

Denotes both the set of artifacts that constitute the unit of component implementation, and the definition of this aggregate entity.

Consumer

An event sink.

Container

Containers provide the run-time execution environment for CORBA component implementations. A container is a framework for integrating transactions, security, events and persistence into a component's behavior at runtime.

Emitter

An event source that can be connected to at most one consumer.

Entity Component

A CORBA component with persistent state, identity which is architecturally visible to clients through a primary key, and behavior, which may be transactional.

Equivalent IDL

The client mappings; that is, mappings of the externally-visible component features for component declarations, or home features for home declarations. Implicitly defined by a component definition in IDL.

Equivalent Interface

The interface that manifests the component's or home's surface features to clients, allowing clients to navigate among the component's facets, and to connect to the component's ports, as defined by the component's or home's equivalent IDL.

Event Sink

A named connection point into which events of a specified type may be pushed.

Event Source

A named connection point that emits events of a specified type to one or more interested event consumers, or to an event channel.

Executor

The programming artifact(s) that supply the behavior of a component or a component home.

Extended Component

Extended components may offer any type of port.

Facet

A distinct named interface provided by the component for client interaction. The primary vehicle through which a component exposes its functional application behavior to clients during normal execution.

Home

A home definition describes an interface for managing instances of a specified component type. A home definition implicitly defines an equivalent interface, which can be described in terms of IDL. A home may be associated with a primary key specification.

Monolithic Executor

An executor consisting of a single artifact.

Multiplex Receptacle

A specialization of a receptacle that allows multiple simultaneous connections.

Primary Key

Primary key values uniquely identify component instances within the scope of the home that manages them.

Port

A surface feature through which clients and other elements of an application environment may interact with a component. The component model supports four basic kinds of ports: facets, receptacles, event sources, event sinks and attributes.

Process Component

A CORBA component with persistent state which is not visible to the client, persistent identity, and behavior, which may be transactional.

Proxy Home

Implements the component home interface specified by a composition definition, but the implementation is not required to be collocated with the container where the components managed by the home are activated.

Publisher

An event source that can be connected to an arbitrary number of consumers, who are said to *subscribe* to the publisher event source.

Receptacle

A named connection point that describes the component's ability to use a reference supplied by some external agent.

Segmented Executor

A set of physically distinct artifacts, a physical partition of the executor. Each segment encapsulates independent state and is capable of being independently activated. Each segment provides at least one facet.

Service Component

A CORBA component with behavior, no state, and no identity.

Session Component

A CORBA component with behavior, transient state, and identity (which is not persistent).

Simplex Receptacle

A specialization of a receptacle that only allows a single connection at a given time.

4.2 Keywords for Requirement statements

The keywords “must,” “must not,” “shall,” “shall not,” “should,” “should not,” and “may” in this International Standard are to be interpreted as described in IETF RFC 2119.

5 Symbols (and abbreviated terms)

For the purposes of this International Standard, the following abbreviations apply:

API — Application Programming Interface
CCM — CORBA Component Model
CIDL — Component Implementation Definition Language
CIF — Component Implementation Framework
CMT — Container-managed Transaction
CORBA — Common Object Request Broker Architecture
COS — Common Object Services
CRUD — Create, Read, Update, Delete
DII — Dynamic Invocation Interface
DTD — Document Type Definition
EJB — Enterprise Java Beans
GIOP — General Inter-ORB Protocol
IDL — Interface Definition Language
IIOP — Internet Inter-ORB Protocol
IR — Interface Repository
JDK — Java Development Kit
JNDI — Java Naming and Directory Interface
JTA — Java Transaction API
MOF — Meta Object Facility
OMG — Object Management Group
ORB — Object Request Broker
PIM — Platform Independent Model (see [MDA])
POA — Portable Object Adapter
PSDL — Persistent State Definition Language
PSM — Platform Specific Model (see [MDA])
RMI — Remote Method Invocation

SECIOP — Secure Inter-ORB Protocol

SMT — Self-managed Transaction

SSL — Secure Sockets Layer

UML — Unified Modeling Language

URI — Uniform Resource Identifier

URL — Uniform Resource Locator

XMI — XML Metadata Interchange

XML — Extensible Markup Language

6 Component Model

6.1 Component Model

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

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.

6.1.1 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 “componentize” 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.

6.1.2 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 component’s ability to use a reference supplied by some external agent.
- **Event sources**, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel.
- **Event sinks**, which are named connection points into which events of a specified type may be pushed.

- **Attributes**, which are named values exposed through accessor and mutator operations. 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.

6.1.3 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 component's *equivalent interface*, that manifests the component's surface features to clients. The equivalent interface allows clients to navigate among the component's facets, and to connect to the component's ports.

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 6.1 illustrates the relationship between the component and its facets.

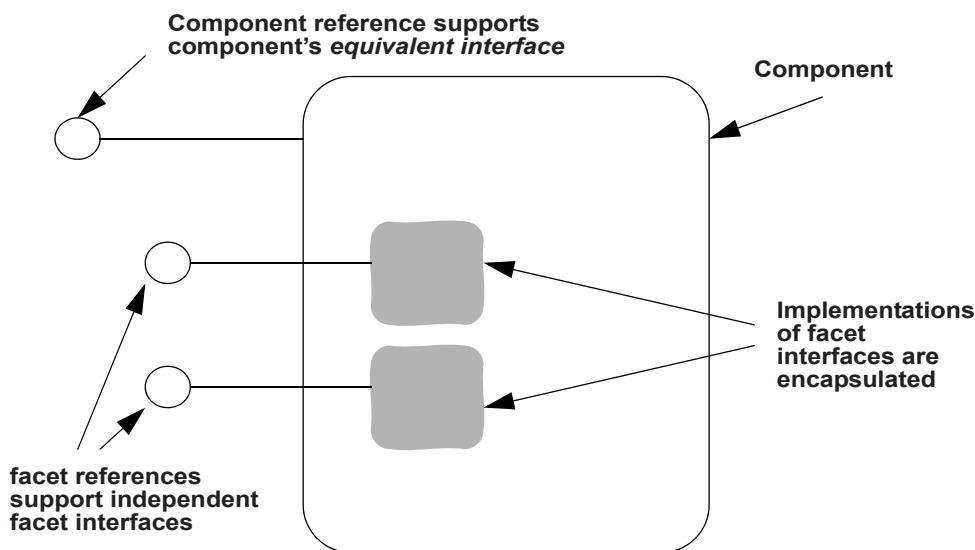


Figure 6.1- Component Interfaces and Facets

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

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

6.1.4 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 component's references. The definition of "same" component instance is ultimately up to the component implementor, in that they may provide a customized implementation of this operation. However, a component framework shall provide standard implementations that constitute *de facto* definitions of "sameness" when they are employed.

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

6.1.5 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.

6.2 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*.

6.3 Component Declaration

6.3.1 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 (Part 1), *OMG IDL Syntax and Semantics* clause, "Component" sub clause 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.

6.3.2 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.

6.3.2.1 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 { ... };**

6.3.2.2 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> { ... };**

Supported interfaces are described in detail in “Supported interfaces” on page 18.

6.3.2.3 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>* { ... }

6.3.2.4 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>* { ... };

6.3.3 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 component's equivalent interface. These declarations and their meanings are described in detail below.

6.4 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.

6.4.1 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 component's facets are described in "Navigation" on page 14. The relationships between the component identity and the facet references, and assumptions regarding facet references, are described in "Provided References and Component Identity" on page 17. 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*.

6.4.2 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 instance's life cycle. A component implementation may, as part of its advertised behavior, return a nil object reference as the result of a **provide_<name>** operation.
- An object reference returned by a **provide_<name>** operation 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 “Provided References and Component Identity” on page 17 and “Navigation” below.

6.4.3 Navigation

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

- A client may navigate from any facet reference to the component that provides the reference via **CORBA::Object::get_component**.
- A client may navigate from the component interface to any facet using the generated **provide_<name>** operations on the 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.

6.4.3.1 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 “**_component**”. 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.

6.4.3.2 Component-specific provide operations

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

6.4.3.3 Navigation interface on the component

As described in “Component Declaration” on page 11 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 component's 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 component's 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 "Provided References and Component Identity" on page 17.

6.4.3.4 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:

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

A client could navigate from a to b as follows:

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

Or, it could navigate directly:

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

6.4.4 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 “same component instance” means. The skeletons generated by the Component Implementation Framework shall provide an implementation of **same_component**, where “same instance” is defined in terms of opaque identity values supplied by the component implementation or the container in the container context. User-supplied implementations can provide different semantics.

If a facet interface inherits the **Navigation** interface, then the **same_component** operation on the provided interface 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.

6.4.5 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> { ... };

The component implementation shall supply implementations of operations defined on supported interfaces. Clients shall be able to widen a reference of the component's 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 component's supported interfaces.

For example, given the following IDL:

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

The AManager interface 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.CCMAbstractObject, 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 light-weight 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 A's home, but doesn't statically know about A's interface or home interface:

```
org.omg.Components.KeylessCCMHome genericHome =  
... // get A's home;  
org.omg.Components.CCMAbstractObject 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.

6.5 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.

6.5.1 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>Connections;
```

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

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

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

6.5.2 Behavior

6.5.2.1 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 the *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 receptacle's **uses** declaration does not include the optional **multiple** keyword, then only a single connection to the receptacle may exist at a given time. If a client invokes a connect operation when a connection already exists, the connection operation 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 receptacle's **uses** declaration includes the optional **multiple** keyword, then multiple connections to the receptacle may exist simultaneously. The component implementation may choose to establish a limit on the number of simultaneous connections allowed. If an invocation of a connect operation attempts to exceed this limit, the operation 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 "Cookie type" on page 22."

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

6.5.2.2 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.

6.5.2.3 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.

6.5.2.4 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.

6.5.3 Receptacles Interface

The **Receptacles** interface provides generic operations for connecting to a component's 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);

        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 receptacle's **uses** statement, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the implementation-defined limit to the number of connections is exceeded, the **ExceededConnectionLimit** exception is raised.
- 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

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 and 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.

6.6 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 http://www.omg.org/technology/documents/formal/notification_service.htm. 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.

6.6.1 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.

6.6.1.1 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.

6.6.1.2 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 part of ISO/IEC 19500 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.

6.6.2 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).

6.6.3 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.

6.6.4 Event Sources—Publishers and Emitters

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

6.6.5 Publisher

6.6.5.1 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:

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

6.6.5.2 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.

6.6.6 Emitters**6.6.6.1 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:

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

6.6.6.2 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 “EventConsumers” to the component type name) is defined in the same scope as the component’s equivalent interface. The module is opened before the equivalent interface definition to provide forward declarations for consumer interfaces. It is re-opened after the equivalent interface definition to define the consumer interfaces.

- The name of a consumer interface is formed by appending the string “Consumer” to the name of the event type. One consumer interface type is implied for each unique event type used in event source and event sink declarations in the component definition.

6.6.7 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.

6.6.7.1 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:

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

6.6.7.2 Event sink operations

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

6.6.8 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);  
        EventConsumerBase 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

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.

6.7 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 home's equivalent interface to contain a set of implicitly defined operations whose signatures are determined by the types of the primary key and the managed component. These operations are specified in "Home definitions with primary keys" on page 35.

6.7.1 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 operations—creation, finding, and destruction. The signatures differ to make the operations specific to the storage type (and, if present, primary key) associated with the home. These two goals—uniformity and type safety—are admittedly conflicting, and the resulting complexity of equivalent home interfaces reflects this conflict. Note that this complexity manifests itself in generated interfaces and their inheritance relationships; the model seen by the client programmer is relatively simple.

6.7.1.1 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);  
};
```