

CORBA 3.0 New Components Chapters

CCM FTF Draft ptc/99-10-04

Document Editor: Jeff Mischkinsky
jeff_mischkinsky@omg.org

OMG TC Document ptc/99-10-04
October 29, 1999

Table of Contents



60	OMG CIDL Syntax and Semantics	9
60.1	Overview	10
60.2	Lexical Conventions	11
60.2.1	Keywords	11
60.3	OMG CIDL Grammar	11
60.4	OMG CIDL Specification	13
60.5	Composition Definition	13
60.5.1	Life cycle category and constraints	14
60.6	Catalog Usage Declaration	15
60.7	Home Executor Definition	15
60.8	Home Implementation Declaration	16
60.9	Storage Home Binding	17
60.10	Home Persistence Declaration	17
60.11	Executor Definition	17
60.12	Segment Definition	18
60.13	Segment Persistence Declaration	19
60.14	Facet Declaration	19
60.15	Feature Delegation Specification	19
60.16	Abstract Storage Home Delegation Specification	21
60.17	Executor Delegation Specification	23
60.18	Abstract Spec Declaration	23
60.19	Proxy Home Declaration	23
60.20	Scoping Rules	24
61	Component Model	25
61.1	Component Model	26



61.1.1	Component levels	26
61.1.2	Ports	27
61.1.3	Components and facets	27
61.1.4	Component identity	28
61.1.5	Component homes	29
61.2	Component Definition	29
61.3	Component Declaration	29
61.3.1	Basic Components	29
61.3.2	Equivalent IDL	30
61.3.3	Component Body	31
61.4	Facets and Navigation	31
61.4.1	Equivalent IDL	31
61.4.2	Semantics of facet references	32
61.4.3	Navigation	32
61.4.4	Provided References and Component Identity	36
61.4.5	Supported interfaces	36
61.5	Receptacles	38
61.5.1	Equivalent IDL	39
61.5.2	Behavior	40
61.5.3	Receptacles interface	42
61.6	Events	44
61.6.1	Event types	45
61.6.2	Integrity of value types contained in anys	45
61.6.3	EventConsumer interface	45
61.6.4	Event service provided by container	46
61.6.5	Event Sources—publishers and emitters	46
61.6.6	Publisher	47
61.6.7	Emitters	49
61.6.8	Module scope of generated event consumer interfaces	50
61.6.9	Event Sinks	50
61.6.10	Events interface	51
61.7	Homes	53
61.7.1	Equivalent interfaces	53
61.7.2	Primary key declarations	56
61.7.3	Explicit operations in home definitions	57
61.7.4	Home inheritance	58
61.7.5	Semantics of home operations	59
61.7.6	CCMHome interface	61
61.7.7	KeylessCCMHome interface	62
61.8	Home Finders	62
61.9	Component Configuration	64
61.9.1	Exclusive configuration and operational life cycle phases	66
61.10	Configuration with attributes	67

61.10.1	Attribute Configurators	67
61.10.2	Factory-based configuration.....	68
61.11 Component Inheritance	70
61.11.1	CCMObject Interface.....	72
61.12	Conformance Requirements.....	73
61.12.1	A Note on Tools.....	75
61.12.2	Changes to Object Services	75
615	CCM Implementation Framework	77
615.1	Introduction	78
615.2	Component Implementation Framework (CIF) architecture	78
615.2.1	Component Implementation Definition Language (CIDL)	78
615.2.2	Component persistence and behavior.....	78
615.2.3	Implementing a CORBA Component	78
615.2.4	Behavioral elements: Executors.....	79
615.2.5	Unit of implementation : Composition	79
615.2.6	Composition structure	81
615.2.7	Compositions with managed storage	87
615.2.8	Relationship between home executor and abstract storage home ..	90
615.2.9	Executor definition.....	103
615.2.10	Proxy homes.....	112
615.2.11	Component object references.....	113
615.3	Language Mappings	115
62	The Container Programming Model.....	117
62.1	Introduction	118
62.1.1	External API Types	120
62.1.2	Container API Type	120
62.1.3	CORBA Usage Model	120
62.1.4	Component Categories.....	121
62.2	The Server Programming Environment	121
62.2.1	Component Containers.....	121
62.2.2	CORBA Usage Model	123
62.2.3	Component Factories	124
62.2.4	Component Activation	124
62.2.5	Servant Lifetime Management.....	124
62.2.6	Transactions	126
62.2.7	Security	128
62.2.8	Events.....	128
62.2.9	Persistence	129
62.2.10	Application Operation Invocation	131



62.2.11	Component Implementations	131
62.2.12	Component Levels	131
62.2.13	Component Categories	132
62.3	Server Programming Interfaces - Basic Components	137
62.3.1	Component Interfaces	137
62.3.2	Interfaces Common to both Container API Types	138
62.3.3	Interfaces Supported by the Session Container API Type	143
62.3.4	Interfaces Supported by the Entity Container API Type	146
62.4	Server Programming Interfaces - Extended Components	149
62.4.1	Interfaces Common to both Container API Types	150
62.4.2	Interfaces Supported by the Session Container API Type	155
62.4.3	Interfaces Supported by the Entity Container API Type	156
62.5	The Client Programming Model	163
62.5.1	Component-aware Clients	164
62.5.2	Component-unaware Clients	168
64	Integrating with Enterprise JavaBeans	171
64.1	Introduction	172
64.2	Enterprise JavaBeans Compatibility Objectives and Requirements	174
64.3	CORBA Component views for EJBs	175
64.3.1	Mapping of EJB to Component IDL definitions	176
64.3.2	Translation of CORBA Component requests into EJB requests	179
64.3.3	CORBA Component view Example	182
64.4	EJB views for CORBA Components	183
64.4.1	Mapping of Component IDL to Enterprise JavaBeans specifications	183
64.4.2	Translation of EJB requests into CORBA Component requests	186
64.4.3	Example	189
64.5	Comparing CCM and EJB	190
64.5.1	The Home Interfaces	191
64.5.2	The Component Interfaces	192
64.5.3	The Callback Interfaces	194
64.5.4	The Context Interfaces	195
64.5.5	The Transaction Interfaces	196
64.5.6	The Metadata Interfaces	197
66	Component Container Architecture	199
66.1	Component Server	200
66.1.1	Component Levels	201
66.1.2	POA Creation	201
66.1.3	Binding the Container to CORBA services	203
66.1.4	Container API Frameworks	203

66.2 Containers Categories.	206
66.2.1 The Empty Container	206
66.2.2 The Service Container	207
66.2.3 The Session Container	212
66.2.4 The Process Container	221
66.2.5 The Entity Container	232
66.2.6 The EJBSession Container	238
66.2.7 The EJBEntity Container	243
66.3 Persistence Integration	249
66.3.1 Container-managed Persistence	250
66.3.2 Self-managed Persistence.	251
66.3.3 Interactions between the Container and the Persistence Provider	251
66.4 Event Management Integration	252
66.4.1 Channel setup.	253
66.4.2 Transmitting an event.	254
66.4.3 Receiving an event.	254
 69 Packaging and Deployment	 257
69.1 Introduction	258
69.2 Component Packaging	258
69.3 Software Package Descriptor	259
69.3.1 A softpkg Descriptor Example.	260
69.3.2 The Software Package Descriptor XML Elements	261
69.4 CORBA Component Descriptor.	273
69.4.1 Component Feature Description.	273
69.4.2 Deployment Information	274
69.4.3 CIDL Compiler Responsibilities	274
69.4.4 CORBA Component Descriptor Example	275
69.4.5 The CORBA Component Descriptor XML Elements	277
69.5 Component Assembly Packaging.	298
69.6 Component Assembly File	298
69.7 Component Assembly Descriptor	298
69.7.1 Component Assembly Descriptor Example.	300
69.7.2 Component Assembly Descriptor XML Elements.	302
69.8 Property File Descriptor	321
69.8.1 Property File Example	321
69.8.2 Property File XML Elements.	322
69.9 Component Deployment.	327
69.9.1 Participants in Deployment	327
69.9.2 ComponentInstallation Interface	330
69.9.3 AssemblyFactory Interface	331
69.9.4 Assembly Interface.	332



69.9.5 Component Entry Points (Component Home Factories)	332
695.1 softpkg.dtd	335
695.2 corbacomponent.dtd	339
695.3 properties.dtd	345
695.4 componentassembly.dtd	346

Note – In this draft all text in black is from the CCM specification. Text in Brown is from the PSS specifications. Editorial modifications to improve readability are in Dark Green.

Note – Last modified - Jeff Mischkinsky

Note – All cross-references need to be fixed.

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

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	60-10
“Lexical Conventions”	60-11
“OMG CIDL Grammar”	60-11
“OMG CIDL Specification”	60-13
“Composition Definition”	60-13
“Catalog Usage Declaration”	60-15
“Home Executor Definition”	60-15

Section Title	Page
“Home Implementation Declaration”	60-16
“Storage Home Binding”	60-17
“Home Persistence Declaration”	60-17
“Executor Definition”	60-17
“Segment Definition”	60-18
“Segment Persistence Declaration”	60-19
“Facet Declaration”	60-19
“Feature Delegation Specification”	60-19
“Abstract Storage Home Delegation Specification”	60-21
“Executor Delegation Specification”	60-23
“Abstract Spec Declaration”	60-23
“Proxy Home Declaration”	60-23

60.1 Overview

The OMG Component Implementation Definition Language (CIDL) is the language used to describe the

OMG CIDL obeys the same lexical rules as OMG IDL, although new keywords are introduced to support concepts specific to component implementation descriptions.

The description of OMG CIDL’s lexical conventions is presented in Section 60.2, “Lexical Conventions,” on page 60-11. A description of OMG IDL preprocessing is presented in Section 3.3, “Preprocessing,” on page 3-12. The scope rules for identifiers in an OMG IDL specification are described in Section 3.18, “CORBA Module,” on page 3-60.

The OMG CIDL grammar is an extension of the OMG IDL grammar. OMG CIDL is a declarative language. The grammar is presented in Section 60.3, “OMG CIDL Grammar,” on page 60-11.

A source file containing interface specifications written in OMG CIDL must have an “.cdl” extension.

The description of OMG CIDL grammar uses the same syntax notation that is used to describe OMG IDL in xxxxx. For reference, Table 60-1 lists the symbols used in this format and their meaning.

Table 60-1 IDL EBNF

Symbol	Meaning
::=	Is defined to be
	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{ }	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

60.2 Lexical Conventions

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

60.2.1 Keywords

The identifiers listed in Table 60-2 are reserved for use as keywords in CIDL, and may not be used otherwise in CIDL, unless escaped with a leading underscore. These are in addition to the ones defined by IDL, which may also not be used otherwise in CIDL, unless escaped with a leading underscore.

Table 60-2 Keywords

bindsTo	delegatesTo	implements	segment	storageHome
catalog	entity	process	service	storedOn
composition	executor	proxy	session	

60.3 OMG CIDL Grammar

The grammar for CIDL is defined by the following BNF productions:

- (1) **<composition>** ::= "composition" <category> <identifier>
"{" <composition_body> "}"
- (2) **<category>** ::= "entity"
| "process"
| "service"
| "session"
- (3) **<composition_body>** ::= [<catalog_use_dcl>] <home_executor_def>
[<proxy_home_def>]
- (4) **<catalog_use_dcl>** ::= "uses" "catalog" "{" <catalog_dcl>+ "}" ";"

- a mandatory home executor definition
- an optional proxy home definition.

60.5.1 Life cycle category and constraints

Certain composition configurations are only valid for certain life cycle categories. Chapter 62, “The Container Programming Model” describes the life cycle-related constraints from the perspective of the container. These constraints map onto corresponding constraints in component and composition definitions. The following lists define the CIDL constructs that are either mandatory or invalid for the designated life cycle category.

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

Table 60-1 Constraints for service and session components

Service and Session	Mandatory	None
	Invalid	abstract storage home bound to home executor: <abstract_storage_home_binding> in home executor body
		component home implemented by home executor specifies a primary key
		component home implemented by home executor specifies explicit finder operations
		segmented executor: <segment_def> in executor body

Table 60-2 Constraints for process components

Process	Mandatory	None
	Invalid	component home implemented by home executor specifies a primary key

Table 60-3 Constraints for entity components

Entity	Mandatory	component home implemented by home executor specifies a primary key
	Invalid	none

60.6 Catalog Usage Declaration

The syntax for a catalog usage declaration is as follows:

- (4) `<catalog_use_dcl> ::= "uses" "catalog" "{" <catalog_dcl>+ "}" ";;"`
- (5) `<catalog_dcl> ::= <catalog_type_spec> <catalog_label>`
- (6) `<catalog_type_spec> ::= <scoped_name>`
- (7) `<catalog_label> ::= <identifier>`

A catalog usage declaration consists of the following elements:

- the keywords **uses** and **catalog**
- a block containing one or more catalog label declarations

A catalog label declaration consists of the following elements:

- a scoped name denoting a previously-defined catalog
- an identifier that denotes a putative catalog of the specified type within the scope of the composition

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

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

60.7 Home Executor Definition

The syntax for a home executor definition is as follows:

- (8) `<home_executor_def> ::= "home" "executor" <identifier>
"{" <home_executor_body> "}" ";;"`
- (9) `<home_executor_body> ::= <home_impl_dcl>
| [<abstract_storage_home_binding>]
| [<stored_on_dcl>]
| <executor_def>
| [<abstract_storage_home_delegation_spec>]
| [<executor_delegation_spec>]
| [<abstract_spec>]`

A home executor definition consists of the following elements:

- the keywords **home** and **executor**
- an identifier that names the home executor definition within the scope of the composition.

- a home executor body.

The home executor body consists of the following elements:

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

The *<identifier>* in the header of the home executor definition is used as the basis for the name of the skeleton artifact generated by the CIF. The specific forms of the executors are defined in language mappings. The general requirements for language mappings of homes executors are defined in Section 60.7, “Home Executor Definition,” on page 60-15.

60.8 Home Implementation Declaration

The syntax of a home implementation declaration is as follows:

- (10) **<home_impl_dcl> ::= “implements” <home_type_name> “;”**
 (11) **<home_type_name> ::= <scoped_name>**

The home implementation declaration consists of the following elements:

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

The home implementation declaration specifies the component home which is to be implemented by the home executor being defined. The generated skeleton must support the home equivalent interface, as defined in Section 61.7.1, “Equivalent interfaces,” on page 61-53. Implementations of orthodox home operations are generated if the life cycle category of the composition is either **entity** or **process** and the home executor specifies an abstract storage home binding, or if the life cycle category of the executor is either **session** or **service**.

The detailed semantics of generated implementations are described in Section 60.8, “Home Implementation Declaration,” on page 60-16.

- and identifier that names the component executor being defined
- an executor body, containing one or more members enclosed in braces

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

The identifier in the executor definition forms the basis of the name of the programming artifact generated as the executor skeleton. The details of executor structure and responsibilities are defined in Section 60.7, “Home Executor Definition,” on page 60-15, and in CIDL language mappings.

60.12 Segment Definition

The syntax for a segment definition is as follows:

- (19) `<segment_def> ::= “segment” <identifier>
“{” <segment_member>+ “}”`
- (20) `<segment_member> ::= <segment_persistence_dcl> “;”
| <facet_dcl> “;”`

A segment definition consists of the following elements:

- the keyword **segment**
- an identifier that names the segment in the scope of the executor definition
- one or more segment members enclosed in braces

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

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

A separate skeleton is generated by the CIF for each segment of a segmented executor. Segments are independently activated. Each segment is assigned a segment identifier, which as a numeric value of type short, by the CIF implementation. The segment identifier is interpreted internally by the generated implementation during activation. Segment identifiers are also used in component identities, as described in Section 62.4.3.1, “Component Identifiers,” on page 62-156. There is no canonical mechanism for assigning segment identifier values (other than the component segment), as the values of segment identifiers does not affect portability or interoperability.

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

The details of segment structure and implementation responsibilities are described in Section 60.12, “Segment Definition,” on page 60-18.

60.13 Segment Persistence Declaration

The syntax for a segment persistence declaration is as follows:

(21) **<segment_persistence_dcl> ::= “storedOn” <abstract_storage_home_name>
“,”**

A segment persistence declaration has the following elements:

- the keyword **storedOn**
- an abstract storage home name

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

The detailed structure of segments, and implementations responsibilities with respect to segment persistence are described in Section 60.13, “Segment Persistence Declaration,” on page 60-19.

60.14 Facet Declaration

The syntax for a facet declaration is as follows:

(22) **<facet_dcl> ::= “provides” “facet” <identifier>
{ “,” <identifier> }***

A facet declaration has the following elements:

- the keywords **provides** and **facet**
- one or more identifiers separated by commas, where each identifier denotes a facet defined by the component type implemented by the composition (i.e., the component type managed by the home which is implemented by the home executor defined in the composition).

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

The detailed structure of segments, and implementations responsibilities with respect to providing facets are described in Section 60.14, “Facet Declaration,” on page 60-19.

60.15 Feature Delegation Specification

The syntax for a feature delegation specification is as follows:

(23) **<feature_delegation_spec> ::= “delegatesTo” “storage”
 <feature_delegation_list>**
 (24) **<feature_delegation_list> ::= “(” <feature_delegation> { “,”
 <feature_delegation> }* “)”**
 (25) **<feature_delegation> ::= <feature_name> “:”
 <storage_member_name>**

(26) **<feature_name> ::= <identifier>**

(27) **<storage_member_name> ::= <identifier>**

A feature delegation specification has the following elements:

- the keywords **delegatesTo**, **abstract** and **storagetype**
- a list of feature delegation specifications, enclosed in parentheses and separated by commas.

A feature delegation specification consists of the following elements:

- an identifier that denotes a stateful feature of the component implemented by the composition
- a colon
- an identifier that denotes a member of the abstract storage type of the abstract storage home specified in the abstract storage home binding in the home executor definition

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

The following constraints regarding feature delegation must be observed:

- Feature delegation specifications may only occur in an executor definition when the home executor specified an abstract storage home binding.
- The type of the storage member specified in a feature delegation must be compatible with the type of the feature. Compatibility, for the purposes of feature delegation is defined in Table 60-4 on page 20.

Table 60-4 Type compatibility for feature delegation purposes

Feature	Storage member type
attribute	must be identical to feature for all types except object reference and valuetype; for object reference and valuetype storage member must be of identical type or base type (direct or indirect)
receptacle (simplex)	must be identical to feature type or base interface (direct or indirect) of feature type

Table 60-4 Type compatibility for feature delegation purposes

Feature	Storage member type
receptacle (multiplex)	sequence of type compatible with receptacle type as defined above
emitter event source	must be identical to feature type or base interface (direct or indirect) of feature type
publisher event source	long*

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

60.16 Abstract Storage Home Delegation Specification

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

```

(28) <abstract_storage_home_delegation_spec> ::= "delegatesTo" "abstract"
      "storagehome" <delegation_list> " ,"
(30)      <delegation_list> ::= "(" <delegation> { " ," <delegation> }* ")"
(31)      <delegation> ::= <operation_name> [ ":" <operation_name> ]
(32)      <operation_name> ::= <identifier>

```

An abstract storage home delegation specification has the following elements:

- the keywords **delegatesTo**, **abstract**, and **storagehome**
- a list of delegation specifications enclosed in parentheses and separated by commas

A delegation specification has the following elements:

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

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

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

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

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

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

- An operation on the home interface may delegate to at most one operation on the abstract storage home interface.
- An operation on the abstract storage home interface may be the target of at most one delegation from the home interface.
- Implicitly defined operations on the home (i.e., orthodox operations) delegate by default to cognate operations on the abstract storage home, as described by Section 61.7.5.1, “Orthodox operations,” on page 61-60. These default delegations may be over-ridden by explicit delegations. If an operation on the abstract storage home that is normally the default target of a delegation appears as the target of an explicit delegation, then the home operation that normally would have delegated to that target by default shall have no generated implementation (unless one is explicitly defined).

The detailed semantics and implementation responsibilities of delegated abstract storage home operations are described in Section 60.16, “Abstract Storage Home Delegation Specification,” on page 60-21.

60.17 *Executor Delegation Specification*

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

(29) **<executor_delegation_spec> ::= “delegatesTo” “executor”
<delegation_list> “;”**

An executor delegation specification consists of the following elements;

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

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

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

The CIF shall generate an implementation of the home operation that delegates to the defined operation on the executor. The detailed semantics and implementation responsibilities are described in Section 60.17, “Executor Delegation Specification,” on page 60-23.

60.18 *Abstract Spec Declaration*

The syntax for an abstract spec has the following form:

(33) **<abstract_spec> ::= “abstract” <operation_list> “;”**
 (34) **<operation_list> ::= “(” <operation_name>
 { “,” <operation_name> }* “)”**

Note – Description mmissing in submitted text

60.19 *Proxy Home Declaration*

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

```

(35) <proxy_home_def> ::= “proxy” “home” <identifier>
    “{” <proxy_home_member>+ “}” “;”
(36) <proxy_home_member> ::= <home_delegation_spec> “;”
    | <abstract_spec>
(37) <home_delegation_spec> ::= “delegatesTo” “home” <delegation_list>

```

Note – Description missing in submitted text

60.20 *Scoping Rules*

Note – Here we need to state the scoping rules that apply to the CIDL extensions, i.e. which constructs introduce a new scope, etc.

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

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

61.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Component Model”	61-26
“Component Definition”	61-29
“Component Declaration”	61-29
“Facets and Navigation”	61-31
“Receptacles”	61-38
“Events”	61-44
“Homes”	61-53
“Home Finders”	61-62
“Component Configuration”	61-64
“Configuration with attributes”	61-67
“Component Inheritance”	61-70
“Conformance Requirements”	61-73

Section Title	Page

Issue – what to do about “paranetical indented comments”

61.1 Component Model

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

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

A component type encapsulates its internal representation and implementation. Although the component specification includes standard frameworks for component implementation, these frameworks, and any assumptions that they might entail, are completely hidden from clients of the component.

61.1.1 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 mapping and integration at this level much easier.

61.1.2 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.

61.1.3 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 61-1 illustrates the relationship between the component and its facets.

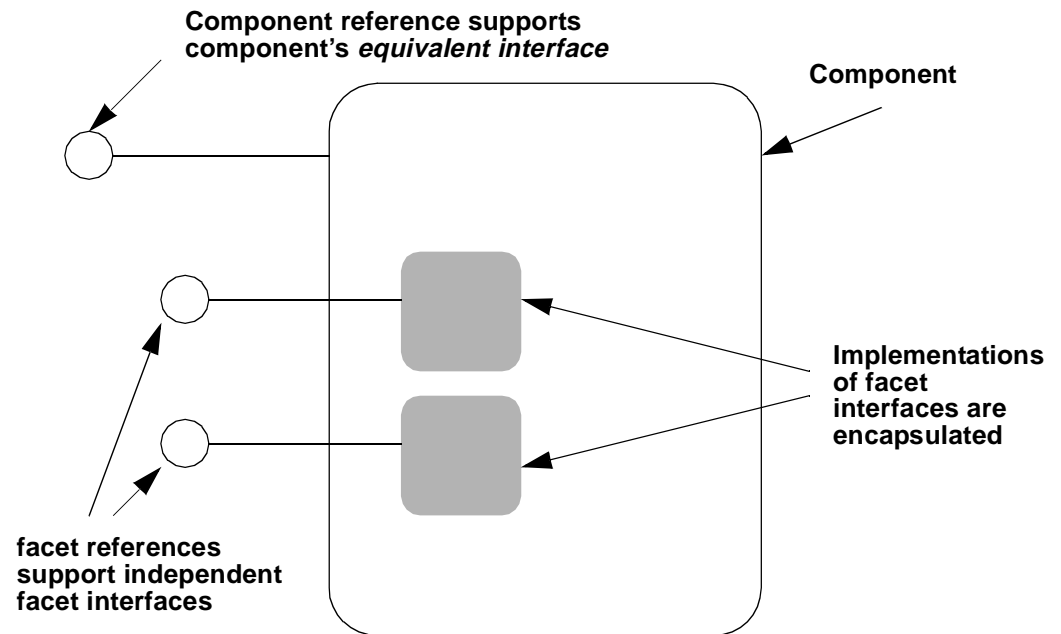


Figure 61-1 Component Interfaces and Facets

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.

61.1.4 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.

61.1.5 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).

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.

61.2 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 <<Chap 3 rrr>>.

61.3 Component Declaration

61.3.1 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 as a restricted version of a **<component_dcl>**. See Section 3.16.1, “Component 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.

61.3.2 Equivalent IDL

The client mappings (i.e., mappings of the externally-visible component features) for component declarations are described in terms of *equivalent IDL*.

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.

61.3.2.1 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 { ... };**

61.3.2.2 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 Section 61.4.5

61.3.2.3 *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> { ... }

61.3.2.4 *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> { ... };

61.3.3 *Component Body*

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

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

61.4 *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.

61.4.1 *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 Section 61.4.3. The relationships between the component identity and the facet references, and assumptions regarding facet references, are described in Section 61.4.4. 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 <<< rrr Chapter ? CIDL>>>.

61.4.2 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 Section 61.4.4 and Section 61.4.3.

61.4.3 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., **provide_all_facets** and **provide_named_facets**)

return multiple references, and can also be used for navigation. When using generic navigation operations on **Navigation**, facets are identified by string values that contain their declared names.

- A client may navigate from a facet interface that derives from the **Navigation** interface directly to any other facet on the same component, using **provide_facet**, **provide_all_facets**, and **provide_named_facets**.
- 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.

61.4.3.1 *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**; see section 15.4.1.2 << rrr>>), the operation name in GIOP request corresponding to **get_component** shall be “**_component**”.

61.4.3.2 *Component-specific provide operations*

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

61.4.3.3 *Navigation interface on the component*

As described in Section 61.3 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 {

    valuetype FacetDescription {
        public CORBA::RepositoryId InterfacelD;
        public FeatureName Name;
    };

    valuetype Facet : FacetDescription {
        public Object ref;
    };

    typedef sequence<Facet> Facets;
    typedef sequence<FacetDescription> FacetDescriptions;

    exception InvalidName { };

    interface Navigation {

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

        FacetDescriptions describe_facets();

        Facets provide_all_facets();

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

        boolean same_component (in Object ref);

    };
};

```

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

provide_facet

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

describe_facets

The **describe_facets** operation returns a sequence containing descriptions of all of the facets provided by the target component. Each description is a value type containing the **RepositoryId** of the facet's interface and the name of the facet, expressed as an unscoped local name relative to the owning component's name scope. The order in which these descriptions occur in the sequence is not specified. A component that does not provide any facets (e.g., a basic component) shall return a sequence of length zero.

provide_all_facets

The **provide_all_facets** operation returns a sequence of value objects, each of which contains the **RepositoryId** of the facet interface and **name** of the facet, along with a reference to the facet. The sequence 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.

provide_named_facets

The **provide_named_facets** operation returns a sequence of described references (identical to the sequence returned by **provide_all_facets**), containing descriptions and references for the facets denoted by the **names** parameter. If any name in the **names** parameter is not a valid name for a provided interface on the component, the operation raises the **InvalidName** exception. The order of values in the returned sequence is not specified. A component that does not provide any facets (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the **InvalidName** exception.

The **same_component** operation on **Navigation** is described in Section 61.4.4.

61.4.3.4 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**, **provide_all_facets**, **provide_named_facets**, and **describe_facets**) are invoked on the facet reference, the operations shall return the same results as if they had been invoked on the component interface that provided the target facet. The skeletons generated by the Component Implementation Framework 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 session {
```

```
        provides foo a;  
        provides bar b;  
    };  
};
```

A client could navigate from a to b as follows:

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

Or, it could navigate directly:

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

61.4.4 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.

61.4.5 Supported interfaces

Issue – dupe of 3.16.2.2 - DELETE description from there

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.CCMLObject, 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.CCMLObject 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.

61.5 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.

61.5.1 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> ( );

```

61.5.2 Behavior

61.5.2.1 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 << rrr Chapter 6 CIDL>>. 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 Section 61.5.2.4, "Cookie type".

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

61.5.2.2 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.

61.5.2.3 *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.

61.5.2.4 *Cookie type*

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

```
module Components {  
    valuetype Cookie {  
        private sequence<octet> CookieValue;  
    };  
};
```

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

Implementations of component-enabled ORBs may employ value type derived from **Cookie**, but any derived cookie types 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.

61.5.3 *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> ConnectedDescriptions;

    interface Receptacles {

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

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

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

    };
};

```

connect

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

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle and it is already connected, then the **AlreadyConnected** exception is raised.
- If the object reference in the **connection** parameter does not support the interface declared in the receptacle's **uses** statement, the **InvalidConnection** exception is raised.

- If the receptacle is a multiplex receptacle and the implementation-defined limit to the number of connections is exceeded, the **ExceededConnectionLimit** exception is raised.
- 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. The following exceptions may be raised:

- If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised.
- If the receptacle is a simplex receptacle there is no current connection, then the **NoConnection** exception is raised.
- If the receptacle is a multiplex receptacle and the cookie value in the **ck** parameter does not denote an existing connection on the receptacle, the **InvalidConnection** exception is raised.
- If the receptacle is a multiplex receptacle and a null value is specified in the **ck** parameter, the **CookieRequired** exception is raised.
- A component that does not have any receptacles (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the **InvalidName** exception.

get_connections

The **get_connections** operation returns a sequence of **ConnectionDescription** structs. Each struct contains an object reference connected to the receptacle named in the **name** parameter, and a cookie value that denotes the connection. If the **name** parameter does not specify a valid receptacle name, then the **InvalidName** exception is raised. A component that does not have any receptacles (e.g., a basic component) will have no valid **name** parameter to this operation and thus shall always raise the **InvalidName** exception.

61.6 Events

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

61.6.1 Event types

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

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

Applications derive specific concrete event types from this base type.

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 and is described in Section 66.4, “Event Management Integration,” on page 66-252.

61.6.2 Integrity of value types contained in anys

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

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.

Issue - Above should be in valuetype chapter

61.6.3 EventConsumer interface

The component event model is a push model. The basic mechanics of this push model are defined by consumer interfaces. Event sources hold references to consumer interfaces and invoke various forms of push operations to send events.

Component event sources hold references to consumer interfaces and push to them. Component event sinks provide consumer references, into which other entities (e.g., channels, clients, other component event sources) push events.

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

```

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

```

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

The **push_event** operation pushes the event denoted by the **evt** parameter to the consumer. The consumer may choose to constrain the type of event it accepts. If the actual type of the **evt** parameter is not acceptable to the consumer, the **BadEventType** exception 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).

61.6.4 Event service provided by container

Container implementations provide event services to components and their clients. Component implementations obtain event services from the container during initialization, and mediate client access to those event services. The container implementation is free to provide any mechanism that supports the required semantics. The container is responsible for configuring the mechanism and determining the specific quality of service and routing policies to be employed when delivering events. More detail is defined in Chapter 66, “Component Container Architecture”, specifically Section 66.4, “Event Management Integration,” on page 66-252.

61.6.5 Event Sources—publishers and emitters

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

61.6.6 Publisher

61.6.6.1 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> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

    interface <component_name> : Components::CCMObject {

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

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

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

```

61.6.6.2 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 a **InvalidConnection** exception.

61.6.7 Emitters

61.6.7.1 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> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

    interface <component_name> : Components::CCMObject {

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

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

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

61.6.7.2 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.

61.6.8 Module scope of generated event consumer interfaces

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

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

61.6.9 Event Sinks

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

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

61.6.9.1 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> {
    module <component_name>EventConsumers {
        interface <event_type>Consumer;
    };

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

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

```

61.6.9.2 Event sink operations

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

61.6.10 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 { };

    interface Events {
        EventConsumerBase get_consumer (in FeatureName sink_name)
            raises (InvalidName);
        Cookie subscribe (in FeatureName publisher_name,
                           in EventConsumerBase subscriber)
            raises (InvalidName);
        void unsubscribe (in FeatureName publisher_name,
                           in Cookie ck)
            raises (InvalidName, InvalidConnection);
        void connect_consumer (in FeatureName emitter_name,
                               in EventConsumerBase consumer)
            raises (InvalidName, AlreadyConnected);
        EventConsumerBase
        disconnect_consumer (in FeatureName source_name)
            raises (InvalidName, NoConnection);

    };

};

```

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.

unsubscribe

The **unsubscribe** operation disassociates the subscriber associated with **ck** parameter with the event source specified by the **publisher_name** parameter. 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. 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.

61.7 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 Section 61.7.1.2, "Home definitions with primary keys".

61.7.1 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.

61.7.1.1 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();
};
```

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

where *<equivalent_explicit_operations>* are the operations defined in the home declaration (*<explicit_operations>*), with factory and finder operations transformed to their equivalent operations, as described in Section 61.7.3, “Explicit operations in home definitions”.

create

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

61.7.1.2 Home definitions with primary keys

Given a home of the following form:

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

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

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

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

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

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

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

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

where *<equivalent_explicit_operations>* are the operations defined in the home declaration (*<explicit_operations>*), with factory and finder operations transformed to their equivalent operations, as described in Section 61.7.3, “Explicit operations in home definitions.

create

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

find_by_primary_key

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

remove

This operation removes the component identified by the specified key value. Subsequent requests to any of the component's facets shall raise a **OBJECT_NOT_EXIST** system exception. If the specified key value does not identify an existing component managed by the home, the operation shall raise an **UnknownKeyValue** exception. If the key value was not a well-formed, legal value, the operation shall raise the **InvalidKey** exception.

61.7.2 Primary key declarations

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

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

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

61.7.2.1 Primary key type constraints

Primary key and types are subject to the following constraints:

- A primary key type must be a value type derived from **Components::PrimaryKeyBase**.
- A primary key type must be a concrete type with at least one public state member.
- A primary key type may not contain private state members.
- A primary key type may not contain any members whose type is a CORBA interface reference type, including references for interfaces, abstract interfaces, and local interfaces.

- These constraints apply recursively to the types of all of the members, i.e., members which are structs, unions, value types, sequences or arrays may not contain interface reference types. If the type of a member is a value type or contains a value type, it must meet all of the above constraints.

61.7.2.2 *PrimaryKeyBase*

The base type for all primary keys is the abstract value type

Components::PrimaryKeyBase. The definition of **PrimaryKeyBase** is as follows:

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

61.7.3 *Explicit operations in home definitions*

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

61.7.3.1 *Factory operations*

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

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

The equivalent operation on the explicit interface is as follows:

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

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

61.7.3.2 *Finder operations*

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

```

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

```

The equivalent operation on the explicit interface is as follows:

```

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

```

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

61.7.3.3 Miscellaneous exports

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

61.7.4 Home inheritance

Given a derived home definition of the following form:

```

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

```

The resulting explicit interface has the following form:

```

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

```

where *<equivalent_explicit_operations>* are the operations defined in the home declaration (*<explicit_operations>*), with factory and finder operations transformed to their equivalent operations, as described in Section 61.7.3, "Explicit operations in home definitions. The forms of the implicit and equivalent interfaces are identical to the corresponding forms for non-derived storage homes, determined by the presence or absence of a primary key specification.

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

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

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

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

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

61.7.5 Semantics of home operations

Operations in home interfaces fall into two categories:

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

Orthodox operations include the following:

- Operations defined on **CCMHome** and **KeylessCCMHome**.
- Operations that appear on the implicit interface for any home.

Heterodox operations include the following:

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

61.7.5.1 *Orthodox operations*

Because of the inheritance structure described in Section 61.7.4, “Home inheritance”, problems relating to polymorphism in orthodox operations are limited. For the purposes of determining key uniqueness and mapping key values to components in orthodox operations, equality of value types (given the constraints on primary key types specified in Section 61.7.2.1, “Primary key type constraints”) are defined as follows:

- Only the state of the primary key type specified in the home definition (which is also the actual parameter type in operations using primary keys) shall be used for the purposes of determining equality. If the type of the actual parameter to the operation is more derived than the formal type, the behavior of the underlying implementation of the operation shall be as if the value were truncated to the formal type before comparison. This applies to all value types that may be contained in the closure of the membership graph of the actual parameter value, i.e., if the type of a member of the actual parameter value is a value type, only the state that constitutes the member’s declared type is compared for equality.
- Two values are equal if their types are precisely equivalent and the values of all of their public state members are equal. This applies recursively to members which are value types.
- If the values being compared constitute a graph of values, the two values are equal only if the graphs are isomorphic.
- Union members are equal if both the discriminator values and the values of the union member denoted by the discriminator are precisely equal.
- Members which are sequences or arrays are considered equal if all of their members are precisely equal, where order is significant.

61.7.5.2 *Heterodox operations*

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

may be the intended behavior of the designer that actual parameter values of either K or K' are legitimate, and the implementation of the operation determines what the appropriate semantics of operation are with respect to key equality.

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

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

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

61.7.6 CCMHome interface

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

```
module Components {
    interface CCMHome {
        CORBA::IObject get_component_def();
        CORBA::IObject get_home_def ();
        void remove_component ( in CCMObject comp);
    };
};
```

get_component_def

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

get_home_def

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

remove_component

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

61.7.7 KeylessCCMHome interface

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

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

create_component

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

61.8 Home Finders

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

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

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

Issue – The string, “ComponentHomeFinder” is added to the list of valid ObjectID values. Make sure and DELETE

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

```

module Components {

    exception HomeNotFound { };

    interface HomeFinder {
        CCMHome find_home_by_component_type (
            in CORBA::RepositoryId comp_repid) raises (HomeNotFound);
        CCMHome find_home_by_home_type (
            in CORBA::RepositoryId home_repid) raises (HomeNotFound);
        CCMHome find_home_by_name (
            in string home_name) raises (HomeNotFound);
    };
};

```

find_home_by_component_type

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

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

find_home_by_home_type

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

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

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

find_home_by_name

Issue – replace with proper INS ref - “param shall contain...”

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

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

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

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

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

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

61.9 Component Configuration

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

Issue – How much and where of this do we keep

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

The configuration framework is designed to provide the following capabilities:

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

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

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

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

client that will use it operationally. For this reason, the CORBA component model provides general guidelines and optional mechanisms that may be employed to characterize configuration operations, but does not attempt to define a strict separation of configuration and operational behaviors.

61.9.1 Exclusive configuration and operational life cycle phases

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

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

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

61.9.1.1 Enforcing exclusion of configuration and operation

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

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

The component implementation framework provides standard mechanisms to support disabling operations during configuration or operation. Certain operations are implemented by the component implementation framework (see Chapter 615, “CCM Implementation Framework”), and may not be disabled.

61.10 Configuration with attributes

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

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

61.10.1 Attribute Configurators

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

61.10.1.1 The Configurator interface

The following interface is supported by all configurators:

```
module Components {

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

configure

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

61.10.1.2 The StandardConfigurator interface

The **StandardConfigurator** has the following definition:

```

module Components {

    valuetype ConfigValue {
        FeatureName name;
        any value;
    };

    typedef sequence<ConfigValue> ConfigValues;

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

```

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

set_configuration

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

61.10.2 Factory-based configuration

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

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

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

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

61.10.2.1 *HomeConfiguration interface*

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

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

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

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

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

```
module Components {

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

};
```

set_configurator

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

set_configuration_values

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

complete_component_configuration

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

disable_home_configuration

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

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

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

61.11 Component Inheritance

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

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

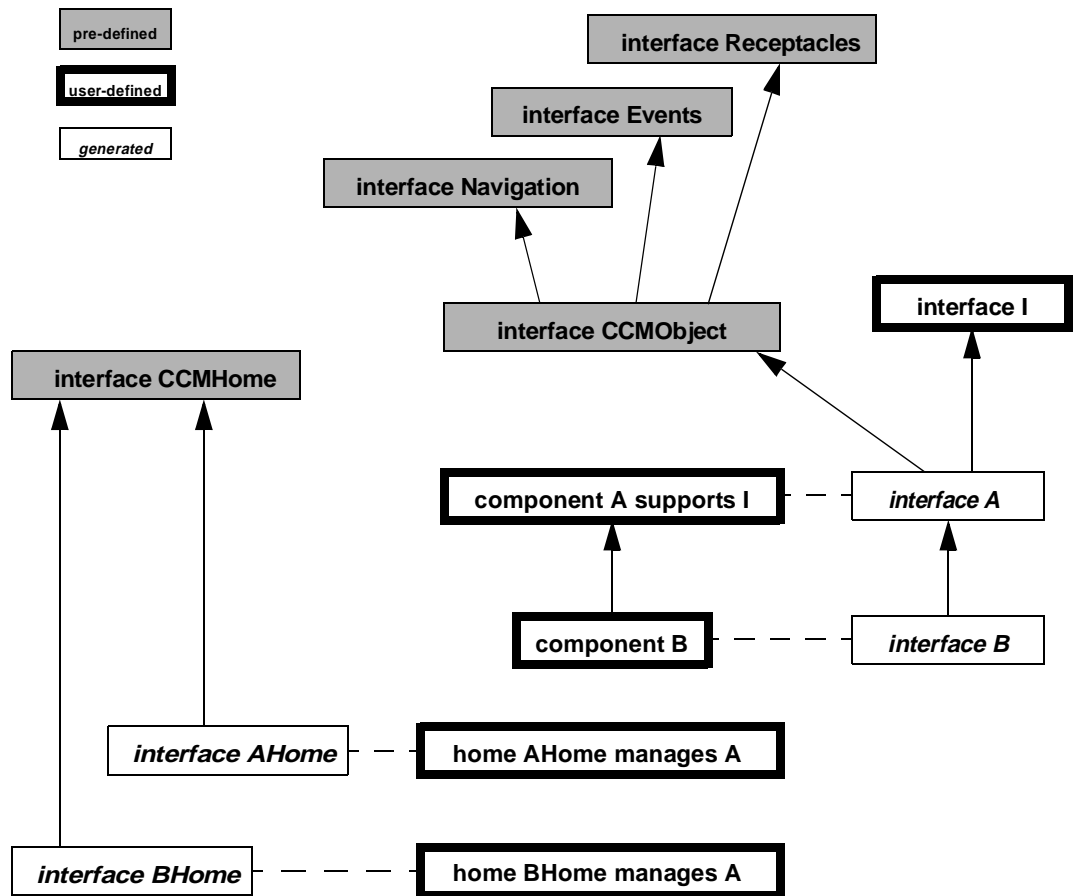


Figure 61-1 Component inheritance and related interface inheritance

61.11.1 CCMObject Interface

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

module Components

```

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

```

```

};

```


get_component_def

This operation returns an **IObject** reference to the component definition in the Interface Repository. The interface repository representation of a component is defined in Volume III of this specification. In strongly typed languages, the **IObject** returned must be narrowed to **IR::ComponentDef** before use.

get_ccm_home

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

get_primary_key

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

configuration_complete

This operation is called by a configurator to indicate that the initial component configuration has completed. If the component determines that it is not sufficiently configured to allow normal client access, it raises the **InvalidConfiguration** exception. The component configuration process is described in Section 61.9.

remove

This operation is called when a component is about to be destroyed. The component can perform any cleanup processing required (e.g. releasing resources) prior to its destruction.

61.12 Conformance Requirements

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

The following conformance points are defined:

1. A CORBA COS vendor shall provide the relevant changes to the Lifecycle, Transaction, and Security Services identified in the following Section 61.12.2, "Changes to Object Services," on page 61-75.
2. A CORBA ORB vendor need not provide implementations of Components aside from the changes made to the Core to support components. Conversely a CORBA Component vendor need not be a CORBA ORB vendor.
3. A CORBA Component vendor shall provide a conforming implementation of the Basic Level of CORBA Components.
4. A CORBA Component vendor may provide a conforming implementation of the Extended Level of CORBA Components.

5. In order to be conformant at the Basic level a non-Java product shall implement (at a minimum) the following:
 - the IDL extensions and generation rules to support the client and server side component model for basic level components.
 - CIDL. The multiple segment feature of CIDL (Section 60.12, “Segment Definition,” on page 60-18) need not be supported for basic components.
 - a container for hosting basic level CORBA components.
 - the XML deployment descriptors and associated zip files for basic components in the format defined in Section 69.1, “Introduction,” on page 69-258.

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

6. In order to be conformant at the Basic level a Java product shall implement (at a minimum):
 - EJB1.1, including support for the EJB 1.1 XML DTD,
 - the java to IDL mapping, also known as RMI/IIOP
 - EJB to IDL mapping as defined in Section 64.3.2, “Translation of CORBA Component requests into EJB requests,” on page 64-179.

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

7. In order to be conformant at the extended level, a product shall implement (at a minimum) the requirements needed to achieve Basic PLUS
 - IDL extensions to support the client and server side component model for extended level components
 - A container for hosting extended level CORBA components.
 - the XML deployment descriptors and associated zip files for basic and enhanced level components in the format defined in Section 69.1, “Introduction,” on page 69-258.

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

8. A CORBA Component vendor may optionally support EJB clients interacting with CORBA Components, by implementing the IDL to EJB mapping as defined in Section 64.4.2, “Translation of EJB requests into CORBA Component requests,” on page 64-186.
9. This specification includes extensions to IDL, in the form of new keywords and grammar. Although a CORBA ORB vendor need not be a CORBA Component vendor, and vice-versa, it is important to maintain IDL as a single language. To this end, all compliant products of any conformance points above shall be able to parse any valid IDL definitions. However, it is permitted to raise errors, or to ignore, those parts of the grammar that relate to another conformance point.

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

61.12.1 A Note on Tools

Component implementations are expected to be supported by tools. It is not possible to define conformance points for tools, since a particular tool may only support part of the component development and deployment life-cycle. Hence a suite of tools may be needed. The Component architecture contains a number of definitions that are relevant to tools, including zip files and XML formats, as well as IDL interfaces for customization and installation. Although it cannot be enforced, tools are expected to conform to the relevant areas with which they are dealing. For example, a tool that generates implementations for a particular platform is expected to generate XML according to the `<implementation>` clauses in the DTD defined in 10.

61.12.2 Changes to Object Services

61.12.2.1 Life Cycle Service

To support the factory design pattern for creating a component instance and to allow the server, rather than a client, to select from a group of functionally equivalent factories based on load or other server-side visible criteria, the following operation is added to the **FactoryFinder** interface of the **CosLifeCycle** module:

```
module CosLifeCycle {
    interface FactoryFinder {
        Factory find_factory (in Key factory_key) raises (noFactory);
    };
};
```

The parameters of the above operation are as defined by **CosLifeCycle** with the following clarifications:

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

61.12.2.2 Transaction Service

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

- **CosTransactions::Current**

61.12.2.3 Security Service

The following CORBA Security interfaces are changed to local interfaces:

- **SecurityLevel1::Current**
- **SecurityLevel2::PrincipalAuthenticator**

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

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

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

615.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Introduction”	615-78
“Language Mappings”	615-78
“Language Mappings”	615-115

615.1 Introduction

The Component Implementation Framework (CIF) defines the programming model for constructing component implementations. Implementations of components and component homes are described in CIDL. See Chapter 60, “OMG CIDL Syntax and Semantics” for the definition and syntax. The CIF uses CIDL descriptions to generate programming skeletons that automate many of the basic behaviors of components, including navigation, identity inquiries, activation, state management, lifecycle management, and so on.

615.2 Component Implementation Framework (CIF) architecture

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

615.2.1 Component Implementation Definition Language (CIDL)

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

615.2.2 Component persistence and behavior

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

Issue – Above statement needs to be clarified.

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

This chapter addresses the elements of the CIF that pertain to the implementation of a component’s behavior.

615.2.3 Implementing a CORBA Component

The remainder of section 615.2 provides an overview of the concepts involved in building component implementations. It is intended to provide a high-level description that will serve as a framework for understanding the more formal descriptions that

follow in subsequent sections. While the information in this section is normative (with the exception of italicized, indented rationale), it is not intended to be a complete or precise specification of the CIF, or all of the possible design options from which a component implementor may choose.

615.2.4 Behavioral elements: *Executors*

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

We chose to use the word executor rather than servant to avoid confusion with POA servants. POA servants, while conceptually similar to executors, are significantly different in detail, and map to different types in programming languages. Executor is pronounced with the accent on the second syllable (e.g. -ZEK-yoo-tor).

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

615.2.5 Unit of implementation : *Composition*

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

A composition definition specifies the following elements:

Component home

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

Abstract Storage home binding

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

Home executor

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

Component executor

A composition specifies an executor definition. The name of the executor definition is used as the name of the programming artifact generated by the CIF as the skeleton of the component executor. The body of the executor definition optionally specifies executor *segments*, which are physical partitions of the executor, encapsulating independent state and capable of being independently activated. Segments are described in Section 615.2.9.1, “Segmented executors”. The executor body may also specify a mapping, or *delegation*, of certain component features (e.g., attributes) to storage members.

Delegation specification

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

Proxy home

A composition may optionally specify a proxy home. The CIF supports the ability to define proxy home implementations, which are not required to be collocated with the container that executes the component implementation managed by the home. In some configurations, proxy homes can provide implementations of home operations without contacting the container that executes the actual home and component implementation. Support for proxy homes is intended to increase the scalability of the CORBA Component Model. The use of proxy homes is completely transparent to component clients and, to a great extent, transparent to component implementations. Proxy home behavior is described in Section 615.2.10.1, “Proxy home delegation”.

615.2.6 Composition structure

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

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

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

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

- the name of the composition
- the life cycle category of the component implementation, either **service**, **session**, **process**, or **entity**, as defined in Section 62.1.4, “Component Categories.
- the home type being implemented (which implicitly identifies the component type being implemented)
- the name of the home executor to be generated
- the name of the component executor skeleton to be generated

A composition definition has the following essential form:

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

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

This is a schematic representation of the minimal form of a composition, which specifies no state management. The structure of the composition specified by this schematic is illustrated in Figure 615-1. Note that the component type itself is not explicitly specified. It is unambiguously implied by the specification of the home type,

as is the relationship between the executor and the component (i.e., that the executor *implements* the component).

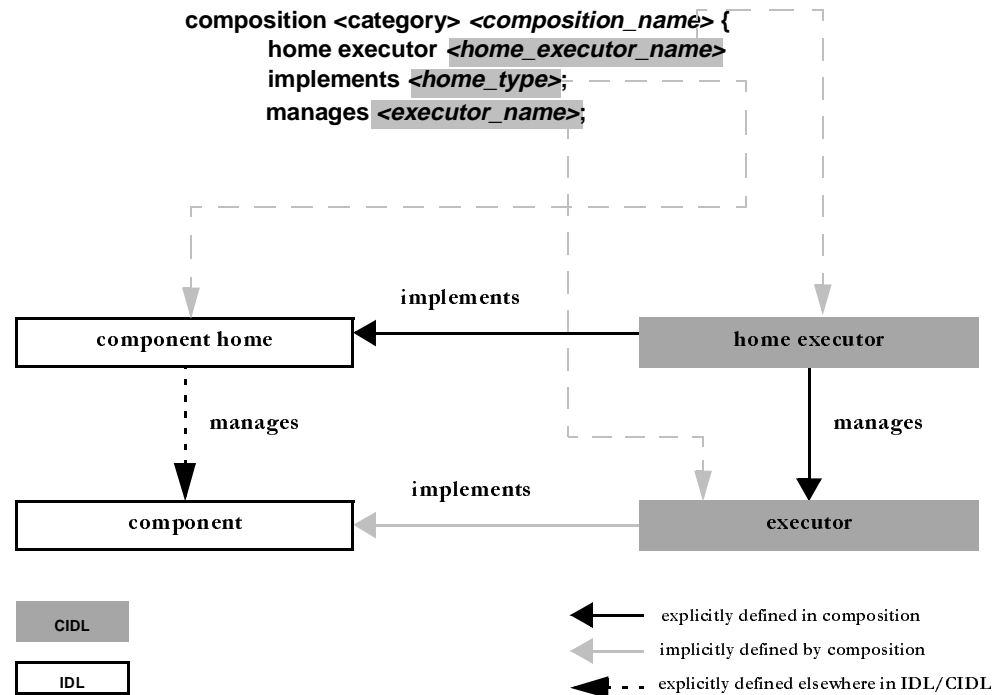


Figure 615-1 Minimal composition structure and relationships

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

Before presenting programming examples, it should be noted that all examples are non-normative illustrations. In particular, the implementations provided in the examples of code that is to be generated by the CIF are merely schematic representations of the intended behaviors; they are by no means indicative of the actual content of a real implementation (e.g., they generally don't include exception handling, testing for validity, etc.).

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

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

```

interface Cat {
    void eat (in Bird lunch);
};
component Toon {
    provides Bird tweety;
    provides Cat sylvester;
};
};

```

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

```

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

module MerryMelodies {

    // this is the composition:

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

```

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

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

- The skeleton for the component executor *ToonSessionImpl*
- The complete implementation of the home executor *ToonTownImpl*

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

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

guage mapping:

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

import org.omg.Components.*;

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

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

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

public interface ToonTownExplicitOperations
extends CCMHomeOperations { }

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

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

The ToonImpl executor skeleton class has the following form:

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

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

```

//

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

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

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

```

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

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

```

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

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

    CCMObject create_component()
    {
        return create();
    }

    void remove_component(CCMObject comp)
    {
    }
}

```

```

    Toon create()
    {
    }
    // and so on...
}
-----

```

The user-provided executor implementation must supply the following:

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

The following example shows one possible implementation strategy:

```

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

public class myToonImpl extends ToonImpl
implements BirdOperations, CatOperations {

    protected long timeFlown;
    protected Bird lastBirdEaten;

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

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

```

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

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

```

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

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

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

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

```

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

615.2.7 Compositions with managed storage

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

- one or more catalogs that provide the storage homes to the composition implementation. Each specified catalog is assigned a alias, or label, that identifies the catalog within the context of the composition.
- an abstract storage home type to which the component home is bound (this implicitly identifies the abstract storage type to which the component itself is bound)
- the life cycle category of the composition must be either **entity** or **process** to support managed storage

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


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

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

The structure of the resulting composition and the relationships between the elements is illustrated in Figure 615-2.

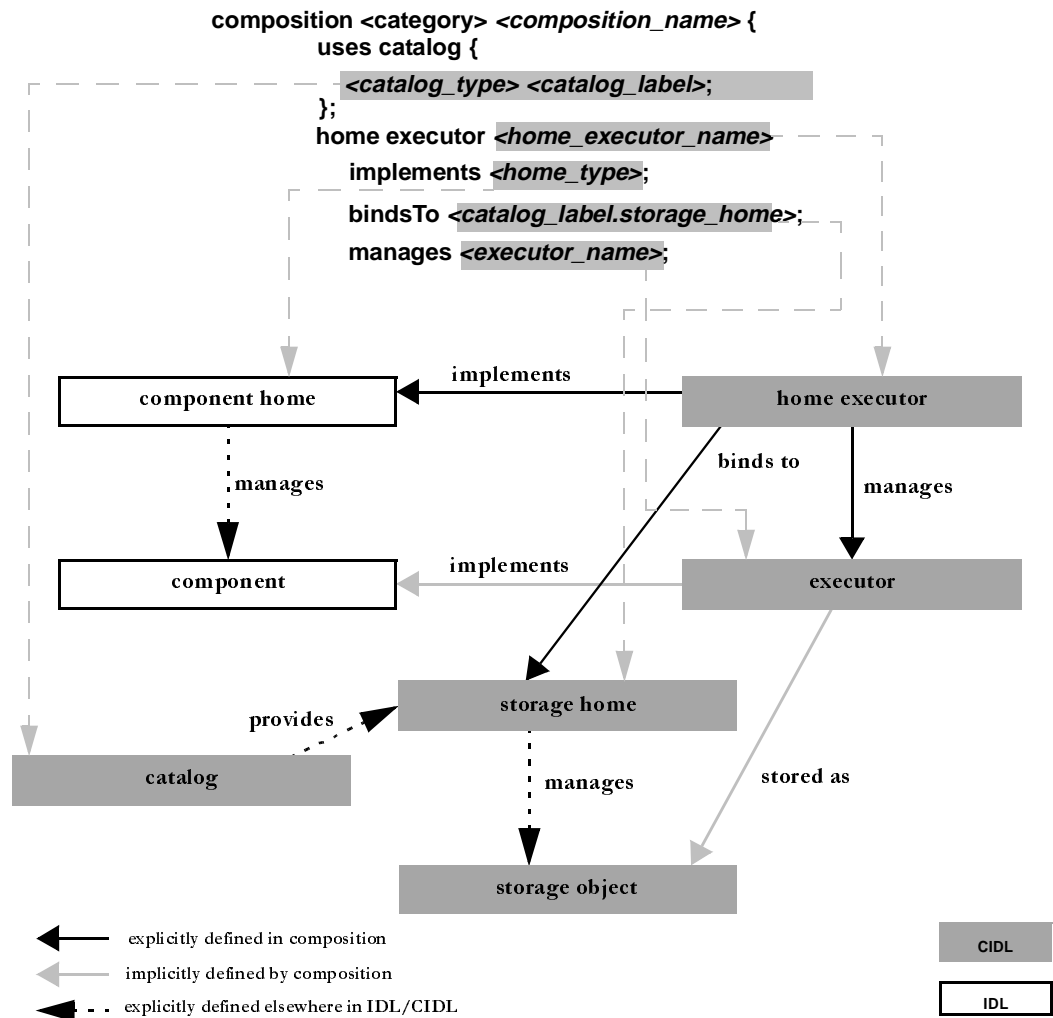


Figure 615-2 Structure of composition with managed storage

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

615.2.8 Relationship between home executor and abstract storage home

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

In general, operations on a home interface provide life cycle management. As described in Section 61.7, when a home definition does not specify a primary key, the resulting equivalent home interface has the following operations:

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

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

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

615.2.8.1 Primary Key Binding

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

For example:

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

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

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

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

615.2.8.2 Implicit delegation of home operations

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

Table 615-1 Delegation of finder operations to finder operations on the bound abstract storagehome

home operation	abstract storagehome operation
<i>component</i> find_by_primary_key (<i>key</i>)	ref< <i>X</i> > find_ref_by_matching_key_name (<i>matching_key</i>)

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

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

- The initial state of the storage object created by the storage home constitutes a valid initial state for the component.
- All of the persistent state of the component is defined on (or reachable from) the storage object whose PID is associated with the component instance.
- The executor is monolithic, not segmented. Home operations can also be delegated to abstract storage homes when the executor is segmented, but the process is slightly more complex, and is discussed in full in Section 615.2.9.1, “Segmented executors,” on page 615-104.

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

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

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

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

    valuetype EpisodeName : Components::PrimaryKeyBase {
        public string name;
```

```

    };
    home ToonTown manages Toon primaryKey EpisodeName {
    };
};
-----

```

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

```

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

module MerryMelodies {

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

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

    catalog ToonCatalog {
        provides ToonStateHome TSHome;
    };

    // this is the composition:

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

```

*In this example, the composition binds the component home **ToonTown** to the abstract storage home **ToonStateHome**, and thus, implicitly binds the component type **Toon** to the abstract storage type **ToonState**. Note that the primary key (if any) in the home must match a key in the abstract storage home. As will be seen later in the CIDL grammar specification, the keyword **entity** in the implementation binding declaration specifies a particular lifecycle model for the resulting implementation.*

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

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

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

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

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

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

import org.omg.Components.*;

... same as previous except for the following:

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

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

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

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

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

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

```
// Example 2
//
// GENERATED FROM CIDL SPECIFICATION:
//
// no explicit operations
public interface ToonStateHome
    extends StorageHomeBase {

    public ToonState
        find_by_episode_name (EpisodeName k);

    public ToonStateRef
        find_ref_by_episode_name (EpisodeName k);
}
```

*The **ToonImpl** executor skeleton class has the following form:*

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

abstract public class ToonImpl
```

```

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

    protected ToonStateIncarnation _state;

    protected ToonImpl() { _state = null; }

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

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

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

```

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

```

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

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

```



```

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

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

    ToonState new_state = _storageHome.create(key);

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

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

// and so on...
}

```

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

```

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

public class myToonImpl extends ToonImpl
implements BirdOperations, CatOperations {

    public myToonImpl() { super(); }

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

```

```

        return (CatOperations) this;
    }
}

```

615.2.8.3 Explicit delegation of home operations

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

```

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

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

```

Delegation to abstract storage home

The **delegatesTo abstract storagehome** declaration specifies a sequence of operation mappings, where each operation mapping specifies the name of an operation on the home, and the name of an operation on the storage home. The signatures of the operations must be compatible, as defined in Section 61.7.4, “Home inheritance,” on page 61-58. Based on this declaration, the CIF generates implementations of the home operations on the home executor that delegate to the specified operations on the abstract storage home.

Delegation to executor

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

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

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

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

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

These precise semantics of and requirements for factory operations delegated to the executor are described in detail in Section 61.7.3.1, “Factory operations,” on page 61-57.

Suppressing generated implementation

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

The following example extends the previous example to illustrate delega-

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

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

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

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

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

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

module MerryMelodies {

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

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

    composition entity ToonImpl {
        uses catalog { ToonCatalog store; };
        home executor ToonTownImpl {
            implements LooneyToons::ToonTown;
            bindsTo store.TSHome;
            manages ToonEntityImpl;
            delegatesTo abstract storagehome
                (arbitrary_operation : do_something);
            delegatesTo executor ( createToon : createToon );
        };
    };
};
-----
```

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

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

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

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

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

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

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

abstract public class ToonImpl
implements LooneyToons.ToonOperations,
ExecutorSegmentBase, PersistentComponent {
    ... same as previous
}
-----
```

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

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

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

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

public class ToonTownImpl
implements LooneyToons.ToonTownOperations,
CCMHome, ExecutorSegmentBase

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

    ...

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

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

    ...

}
```

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

```
-----
// Example 3
//
```

```

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

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

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

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

```

615.2.9 Executor definition

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

- The name of the executor, which is used as the name of the generated executor skeleton
- Optionally, one or more distinct segments, or physical partitions of the executor. Each segment encapsulates independent state and is capable of being independently activated. Each segment also provides at least one facet.
- Optionally, the generation of operation implementations that manage the state of stateful component features (i.e., receptacles, attributes, and event sources) as members of the component incarnation.
- a delegation declaration that describes a correspondence between stateful component features and members of the abstract storage type that incarnates the component. The CIF uses this declaration to generate implementations of the feature-specific operations (e.g., **connect_** and **disconnect_** operations for receptacles, accessors and mutators for attributes) that store the state associated with each specified feature in the storage member indicated on the right hand side of the delegation.

615.2.9.1 Segmented executors

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

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

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

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

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

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

Figure 615-1, and Figure 615-2, illustrate the structure of monolithic and segmented executors, and the relationships between facets, storage objects, and segments. These figures also illustrate the identity information that is embedded in component and facet object references. Component identity information is described in more detail in Section 61.1.4, "Component identity," on page 61-28.

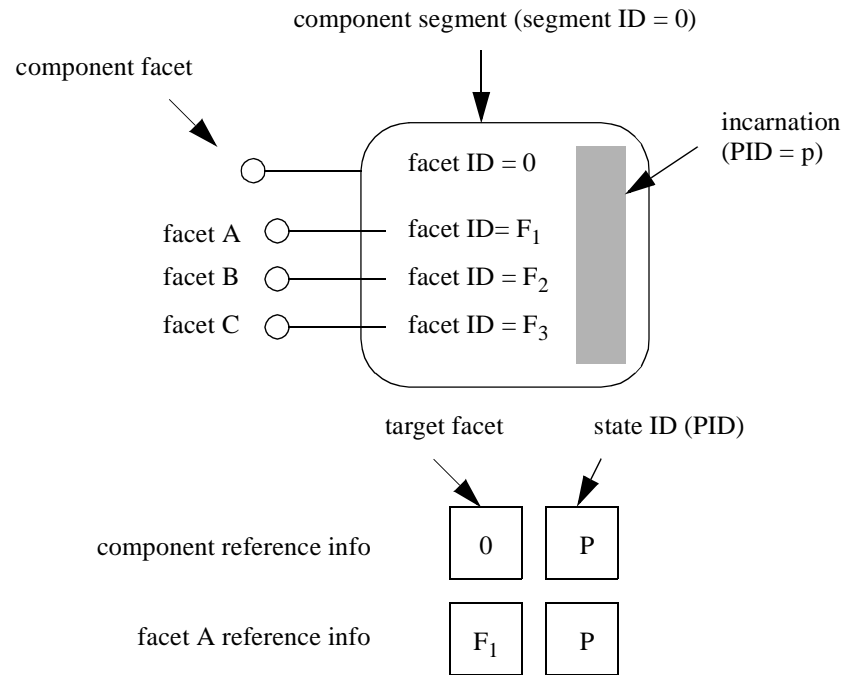


Figure 615-1 Monolithic executor and reference information structure

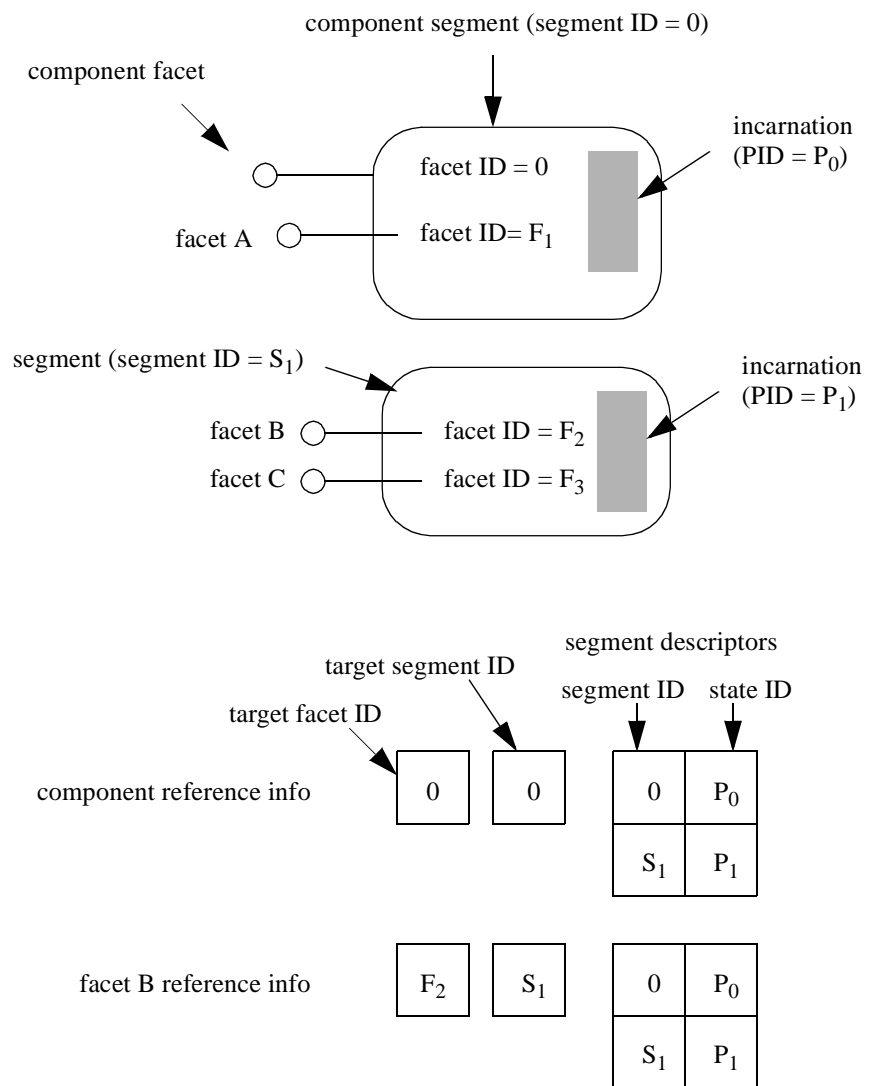


Figure 615-2 Segmented executor and reference information structure

The details of the structure and behavior of segments and requirements for their implementation are specified in Section 615.2.9.1, “Segmented executors,” on page 615-104.

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

```

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

```

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

```

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

module MerryMelodies {

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

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

    abstract storagehome ToonStateHome of ToonState {
        key episode_name;
    };

    abstract storagetype BirdSegState {
        state unsigned long time_flown;
    };

    abstract storagehome BirdSegStateHome of BirdSegState {
        key episode_name;
    };

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

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

```

```

};
};
};
};

```

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

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

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

```

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

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

    protected BirdSegState _state;

    protected BirdSegment() { _state = null; }

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

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

    abstract public BirdOperations
        _get_facet_tweety();
}
-----

```

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

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

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

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

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

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

There are now two segment executors to implement:

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

public class myToonImpl extends ToonImpl
implements CatOperations {
```

```

    public myToonImpl() { super(); }

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

public class myBirdSegImpl extends BirdSegment
implements BirdOperations {

    public myBirdSegImpl() { super(); }

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

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

```

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

```

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

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

    ExecutorSegmentBase
    create_executor_segment (int segid) {

        // case discriminator values are constants
        // generated on the executor segment classes
    }
}

```

```

switch (segid) {
case ToonImpl._segment_id_value :
    return new myToonImpl();
case BirdSegment._segment_id_value :
    return new myBirdSegImpl();
default
    ... raise an exception
}
}
...
}
-----

```

615.2.9.2 Delegation of feature state

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

```

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

```

The type of the storage member must be compatible with the type associated with the feature, as defined in Chapter 61, “Component Model”. In the case of attributes, the CIF-generated implementations of accessors and mutators retrieve and store the attribute value in the specified storage member. The executor programming model allows implementors to intercept invocations of the generated accessor and mutator invocations and replace or extend their behaviors. In the case of receptacles and event sources, the implementations of the **connect_<receptacle_name>**, **disconnect_<receptacle_name>**, **connect_<source_name>**,

disconnect_<source_name>, **subscribe_<source_name>** and **unsubscribe_<source_name>** operations store the connected object references in the specified members of the storage object that incarnates the component.

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

615.2.10 Proxy homes

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

Proxy homes are, in essence, remote projections of the actual home implementation, which is always collocated with the executing component implementation. A proxy home may be able to implement some subset (or potentially, all) of the operations defined on the component home without contacting the actual home implementation. Operations that cannot be locally implemented by the proxy home are delegated to the actual home. The run-time implementation of the CIF (including the supporting infrastructure of the container and the home finder) is responsible for maintaining the associations between proxy homes and the actual home they represent. The container provides an interface for registering proxy homes, described in Section 62.4.1.3, “The ProxyHomeRegistration Interface,” on page 62-152.

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

```
composition <category> <composition_name> {
    ...
    home executor <home_executor_name> {
        implements <home_type> ;
        bindsTo
        <catalog_label.abstract_storage_home>;
        ...
    };
    proxy home <proxy_executor_name> {
        delegatesTo home ( <home_op_0>, <home_op_1>,
        ... );
        abstract (<home_op_2>, <home_op_3>, ...);
    };
};
```

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

The operation delegations specified in the actual home executor definition are also acquired by the proxy home, but certain delegations are transformed according to rules specified in Section 615.2.10.1.

615.2.10.1 *Proxy home delegation*

For proxy homes in compositions that specify managed state, the CIF assumes that the proxy home has connectivity to the same persistent store as the actual home. Based on this assumption, the default implementations of orthodox operations on the proxy home executor are delegated directly to the storage home, precisely as they are in the actual home executor. In general, other operations are delegated to the actual home, by default, although the specific rules for determining the implementation of proxy home operations are somewhat more involved, and are described completely in Section 615.2.3, “Implementing a CORBA Component,” on page 615-78.

615.2.11 *Component object references*

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

The **Entity2Context** interface, described in Section 62.4.3.7, “The Entity2Context Interface,” on page 62-162 is used by the component implementation to provide this information to the container, with which the container creates the object references for the component and its facets. The **ComponentId** interface encapsulates the component reference information. Examples 2, 3, and 4 in the previous sections illustrate the use of the **Entity2Context** and **ComponentId** interfaces to create object references. Figure 615-1, and Figure 615-2, illustrate the structure of the information encapsulated in **ComponentId**, and its relationship to executor structure.

615.2.11.1 *Facet identifiers*

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

615.2.11.2 *Segment identifiers*

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

the appropriate segment executor. The generated implementations of segment executor skeletons define symbolic constants to assist the component implementor in this mapping.

615.2.11.3 *State identifiers*

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

615.2.11.4 *Monolithic reference information*

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

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

615.2.11.5 *Segmented reference information*

The reference information for segmented executors consists of the following:

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

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

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

615.2.11.6 Component identity

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

615.3 Language Mappings

Issue – To be provided

The Container Programming Model

62

This chapter describes the CORBA component **container programming model**.

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

62.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Introduction”	62-118
“The Server Programming Environment”	62-121
“Server Programming Interfaces - Basic Components”	62-137
“Server Programming Interfaces - Extended Components”	62-149
“The Client Programming Model”	62-163

Section Title	Page

The container is the server's runtime environment for a CORBA component implementation. This environment is implemented by a deployment platform such as an application server or a development platform like an IDE. A deployment platform typically provides a robust execution environment designed to support very large numbers of simultaneous users. A development platform would provide enough of a runtime to permit customization of CORBA components prior to deployment but perhaps support a limited number of concurrent users. From the point of view of the CORBA component implementation, such differences are "qualities of service" characteristics and have no effect on the set of interfaces the component implementor can rely on. This chapter is organized as follows:

- Section 62.1 introduces the programming model and defines the elements that comprise it.

The container programming model is an API framework designed to simplify the task of building a CORBA application. Although the framework does not exclude the component developer from using any function currently defined in CORBA, it is intended to be complete enough in itself to support a broad spectrum of applications.

- Section 62.2 describes the programming model the component implementor is to follow.

The programming model identifies the architectural choices which must be made to develop a CORBA component which can be deployed in a container.

- Section 62.3 describes the interfaces seen by the component developer.

These interfaces constitute the contract between the container provider and the component implementor. Together with the client programming interfaces defined in Chapter 61, "Component Model" which can be used by servers as well as clients, they define the server programmer's API.

- Section 62.5 describes the client view of a CORBA component.

The **client programming model** has been described previously (Chapter 61, "Component Model"). This section describes the specific use of CORBA required by a client, which is **NOT** itself a CORBA component, to use a CORBA component written to the server programming model described in Section 62.3.

62.1 Introduction

The container programming model is made up of several elements:

- The **external API types** which define the interfaces available to a component client
- The **container API type** which defines the API framework used by the component developer
- The CORBA usage model which defines the interactions between the container and the rest of CORBA (including the POA, the ORB and the CORBA services)
- The component category which is the combination of the container API type (i.e. the server view) and the external API types (i.e. the client view)

The overall architecture is depicted in Figure 62-1 below::

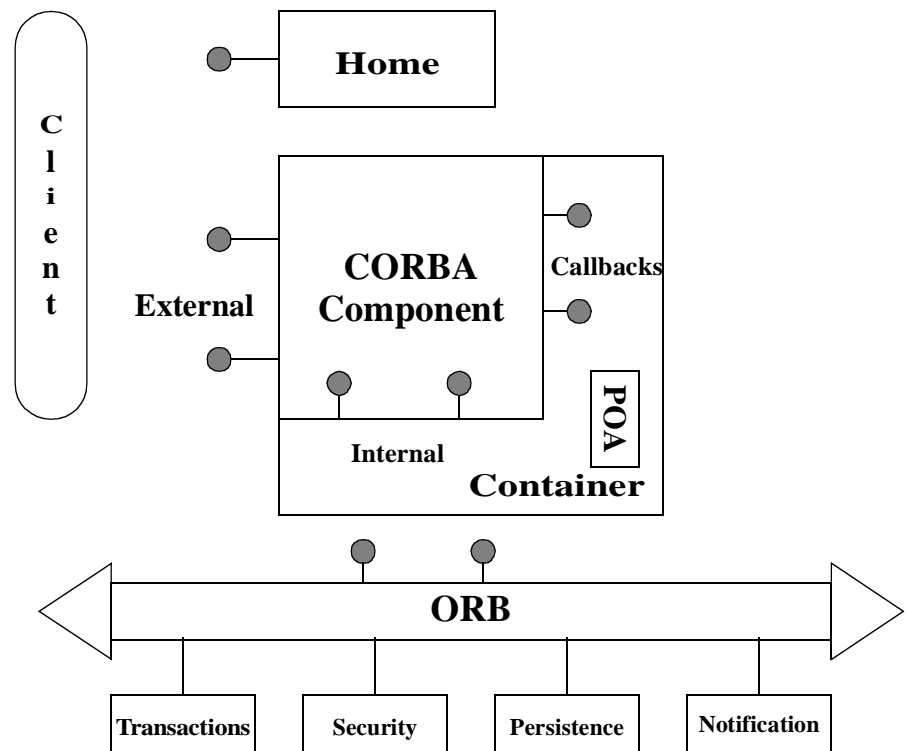


Figure 62-1 The Architecture of the Container Programming Model

The external API types are defined by the component IDL including the home specification. These interfaces are righteous CORBA objects and are stored in the Interface Repository for client use.

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

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

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

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

62.1.1 External API Types

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

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

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

- A home interface with a **primaryKey** declaration supports finders and its client is a **keyfull** client.
- A home interface without a **primaryKey** declaration does not support finders and its client is a **keyless** client. All home types support factory operations.

62.1.2 Container API Type

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

62.1.3 CORBA Usage Model

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

- **stateless** - which uses transient object references in conjunction with a POA servant which can support any **ObjectId**
- **conversational** - which uses transient references in conjunction with a POA servant that is dedicated to a specific **ObjectId**
- **durable** - which uses persistent references in conjunction with a POA servant that is dedicated to a specific **ObjectId**

It should be obvious that the fourth possibility (persistent references with a POA servant that can support any ObjectId) makes no sense and is therefore not included.

62.1.4 Component Categories

The component categories are defined as the valid combinations of external API types, container API type, and CORBA usage model. The following table summarizes the categories and identifies their EJB equivalent:

Table 62-1 Definition of the Component Categories

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

62.2 The Server Programming Environment

The component container provides interfaces to the component. These interfaces support access to CORBA services (transactions, security, notification, and persistence) and to other elements of the component model. This section describes the features of the container which are selected by the deployment descriptor packaged with the component implementation. These features comprise the design decisions to be made in developing a CORBA component. Details of the interfaces provided by the container are provided in Section 62.3, “Server Programming Interfaces - Basic Components.

62.2.1 Component Containers

Containers provide the run-time execution environment for CORBA components. A container is a **framework** for integrating transactions, security, events, and persistence into a component’s behavior at runtime. A container provides the following functions for its component:

- all component instances are created and managed at runtime by its container
- containers provide a standard set of services to a component, enabling the same component to be hosted by different container implementations

Components and homes are deployed into containers with the aid of container specific tools. These tools generate additional programming language and metadata artifacts needed by the container. The tools provide the following services:

- editing the configuration metadata
- editing the deployment metadata
- generating the implementations needed by the containers to support the component

The container framework defines two forms of interfaces:

- **Internal interfaces** - These are locality-constrained interfaces defined as **local interface** types which provide container functions to the CORBA component.

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

- **Callback interfaces** - These are also **local interface** types invoked by the container and implemented by a CORBA component.

*These interfaces provide functions analogous to the **SessionBean** and **EntityBean** interfaces defined by Enterprise JavaBeans.*

This architecture is depicted in Figure 62-1 on page 119.

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

Additional component behavior is controlled by policies specified in the deployment descriptor. This specification defines policies which support POA interactions (CORBA usage model), servant lifetime management, transactions, security, events, and persistence. See the deployment chapter (Chapter 69, “Packaging and Deployment”), specifically Section 69.3, “Software Package Descriptor,” on page 69-259, for details of how container policies are specified.

CORBA containers are designed to be used as Enterprise JavaBeans containers. This allows a CORBA infrastructure to be the foundation of EJB, enabling a more robust implementations of the EJB specification. To support enterprise Beans natively within a CORBA container, the container must support the API frameworks defined by the EJB specification. This architecture is defined in Chapter 64, “Integrating with Enterprise JavaBeans” of this specification.

62.2.2 CORBA Usage Model

The CORBA Component Specification defines a set of **CORBA usage models** which create either **TRANSIENT** or **PERSISTENT** object references and use either a 1:1 or 1:N mapping of **Servant** to **ObjectId**. These CORBA usage models are summarized in Table 62-2 below. A given component implementation shall support one and only one CORBA usage model.

Table 62-2 CORBA Usage Model Definitions

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

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

62.2.2.1 Component References

TRANSIENT objects support only the factory design pattern. They are created by operations on the home interface defined in the **component** declaration.

PERSISTENT objects support either the factory design pattern or the finder design pattern, depending on the component category. **PERSISTENT** objects support **self-managed** or **container-managed** persistence. **PERSISTENT** objects can be used with the CORBA persistent state service or any user-defined persistence mechanism. When the CORBA persistent state service is used, servant management is aligned with the **PersistentId** defined by the CORBA persistent state service and the container supports the transformation of an **ObjectId** to and from a **PersistentId**. A **PersistentId** provides a persistent handle for a class of objects whose permanent state resides in a persistent store (e.g. a database).

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

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

62.2.2.2 Servant to ObjectId Mapping

Component implementations may use either the 1:1 or 1:N mapping of **Servant** to **ObjectId** with **TRANSIENT** references (**stateless** and **conversational** CORBA usage model, respectively) but may use only the 1:1 mapping with **PERSISTENT** references.

- A 1:N mapping allows a **Servant** to be shared among all requests for the same interface and therefore requires the object to be stateless (i.e. it has no identity).
- A 1:1 mapping binds a **Servant** to a specific **ObjectId** for an explicit servant lifetime policy (see Section 62.2.5) and therefore is stateful.

62.2.2.3 Threading Considerations

CORBA components support two threading models: **serialize** and **multithread**. A threading policy of **serialize** means that the component implementation is not thread safe and the container will prevent multiple threads from entering the component simultaneously. A threading policy of **multithread** means that the component is capable of mediating access to its state without container assistance and multiple threads will be allowed to enter the component simultaneously. Threading policy is specified in CIDL.

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

62.2.3 Component Factories

A home is a component factory, responsible for creating instances of all interfaces exported by a component. Factory operations are defined on the home interface using the **factory** declaration. A default factory is automatically defined whose implementation may be generated by tools using the information provided in the **component** IDL. Specialized factories (e.g. factories that accept user-defined input arguments) must be implemented by the component developer. Factory operations are typically invoked by clients but may also be invoked as part of the implementation of the component. A CORBA component implementation can locate its home interface using an interface provided by the container.

62.2.4 Component Activation

CORBA components rely on the automatic activation features of the POA to tailor the behavior of the components using information present in the component's deployment descriptor. Once references have been exported, clients make operation requests on the exported references. These requests are then routed by the ORB to the POA that created the reference and then the component container. This enables the container to control activation and passivation for components, apply policies defined in the component's descriptor, and invoke callback interfaces on the component as necessary.

62.2.5 Servant Lifetime Management

Servants are programming language objects which the POA uses to dispatch operation requests based on the **ObjectId** contained in the object key. The server programming model for CORBA components includes facilities to efficiently manage the memory associated with these programming objects. To implement this sophisticated memory management scheme, the server programmer makes several design choices:

- The container API type must be chosen.
- The CORBA usage model must be chosen.
- A servant lifetime policy is selected. CORBA components support four servant lifetime policies (**method**, **transaction**, **component**, and **container**).
- The designer is required to implement the callback interface associated with his choice.

The servant lifetime policies are defined as follows:

method

The **method** servant lifetime policy causes the container to activate the component on every operation request and to passivate the component when that operation has completed. This limits memory consumption to the duration of an operation request but incurs the cost of activation and passivation most frequently.

transaction

The **transaction** servant lifetime policy causes the container to activate the component on the first operation request within a transaction and leave it active until the transaction completes and which point the component will be passivated. Memory remains allocated for the duration of the transaction.

component

The **component** servant lifetime policy causes the container to activate the component on the first operation request and leave it active until the component implementation requests it to be passivated. After the operation which requests the passivation completes, the component will be passivated by the container. Memory remains allocated until explicit application request.

container

The **container** servant lifetime policy causes the container to activate the component on the first operation request and leave it active until the container determines it needs to be passivated. After the current operation completes, the component will be passivated by the container. Memory remains allocated until the container decides to reclaim it.

The following table (Table 62-3) shows the relationship between the CORBA usage model, the container API type, and the servant lifetime policies.

Table 62-3 Servant Lifetime Policies by Container API Type

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

Table 62-3 Servant Lifetime Policies by Container API Type

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

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

62.2.6 Transactions

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

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

The following table (Table 62-4) summarizes the effect of the various transaction policy declarations and the presence or absence of a client transaction on the transaction which is used to invoke the requested operation on the component.

Table 62-4 Effects of Transaction Policy Declaration

Transaction Attribute	Client Transaction	Component's Transaction
NOT_SUPPORTED	-	-
	T1	-
REQUIRED	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1
REQUIRES_NEW	-	T2
	T1	T2
MANDATORY	-	EXC (TRANSACTION_REQUIRED)

Table 62-4 Effects of Transaction Policy Declaration

Transaction Attribute	Client Transaction	Component's Transaction
	T1	T1
NEVER	-	-
	T1	EXC (INVALID_TRANSACTION)

not_supported

This component does not support transactions. If the client does not provide a current transaction the operation is invoked immediately. If the client provides a current transaction, it is suspended (**CosTransactions::Current::suspend**) before the operation is invoked and resumed (**CosTransactions::Current::resume**) when the operation completes.

required

This component requires a current transaction to execute successfully. If one is supplied by the client, it is used to invoke the operation. If one is not provided by the client, the container starts a transaction (**CosTransactions::Current::begin**) before invoking the operation and attempts to commit the transaction (**CosTransactions::Current::commit**) when the operation completes.

supports

This component will support transactions if one is available. If one is provided by the client, it is used to invoke the operation. If one is not provided by the client, the operation is invoked outside the scope of a transaction.

requires_new

This component requires its own transaction to execute successfully. If no transaction is provided by the client, the container starts one (**CosTransactions::Current::begin**) before invoking the operation and tries to commit it (**CosTransactions::Current::commit**) when the operation completes. If a transaction is provided by the client, it is first suspended (**CosTransactions::Current::suspend**), a new transaction is started (**CosTransactions::Current::begin**), the operation invoked, the component's transaction attempts to commit (**CosTransactions::Current::commit**), and the client's transaction is resumed (**CosTransactions::Current::resume**).

mandatory

The component requires that the client be in a current transaction before this operation is invoked. If the client is in a current transaction, it is used to invoke the operation. If not, the **TRANSACTION_REQUIRED** exception shall be raised.

never

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

62.2.7 Security

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

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

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

62.2.8 Events

CORBA components use a simple subset of the CORBA notification service to emit and consume events. The subset can be characterized by the following attributes:

- Events are represented as **valuetypes** to the component implementor and the component client
- The event data structure is mapped to an **any** in the body of a structured event presented to and received from CORBA notification.
- The fixed portion of the structured event is added to the event data structure by the container on sending and removed from the event data structure when receiving
- Components support two forms of event generation using the push model:
 - a component may be an exclusive supplier of a given type of event.
 - a component may supply events to a shared channel that other CORBA notification users are also utilizing
- A CORBA component consumes both forms of events using the push model.
- Events have transaction and security policies associated with the component's event ports as defined in the deployment descriptor.
- All channel management is implemented by the container, not the component.
- Filters are set administratively by the container, not the component

Because events can be emitted and consumed by clients as well as component implementations, operations for emitting and consuming events are generated from the specifications in component IDL. The container is responsible for mapping these operations to the CORBA notification service to provide a robust event distribution network.

62.2.8.1 *Transaction Policies for Events*

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

normal

A **normal** event policy indicates the event should be generated or consumed outside the scope of a transaction. If a current transaction is active, it is suspended before sending the event or invoking the operation on the proxy object provided by the component.

default

A **default** event policy indicates the event should be generated or consumed regardless of whether a current transaction exists. If a current transaction is active, the operation is transactional. If not, it is non-transactional.

transaction

A **transaction** event policy indicates the event should be generated or consumed within the scope of a transaction. If a current transaction is not active, a new one is initiated before sending the event or invoking the operation on the proxy object provided by the component. The new transaction is committed as soon as the operation is complete.

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

62.2.8.2 *Security Policies for Events*

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

62.2.9 *Persistence*

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

- **container-managed persistence (CMP)** - the component developer simply defines the state which is to be made persistent and the container (in conjunction with generated code) automatically saves and restores state as required.

Container-managed persistence is selected by defining the abstract state associated with a component segment using the state declaration language of the CORBA persistent state service and connecting that state declaration to a component segment using CIDL.

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

Self-managed persistence is selected via CIDL declaration and triggered by the container invoking the callback interfaces (which the component must implement) defined later in this chapter (Section 62.3).

The following table (Table 62-5) summarizes the choices and their required responsibilities:

Table 62-5 Persistence Support for Entity Container API Type

Persistence Support	Persistence Mechanism	Responsibility	Persistence Classes	Callback Interfaces
Container Managed	CORBA	Container	Generated Code	Generated Code
Container Managed	User	Container	Component implements	Generated Code
Self-managed	CORBA	Component	Generated Code	Component implements
Self-managed	User	Component	Component implements	Component implements

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

62.2.9.1 Container-managed Persistence

Container-managed persistence may be accomplished using the CORBA persistent state service or any user-defined persistence mechanism. When the CORBA persistent state service is used, the container manages all interactions with the persistence provider and the component developer need not use the persistence interfaces offered by the container. With container-managed persistence using the CORBA persistent state service, it is possible to provide automatic code generation for the storage factories, finders, and some callback operations.

If container-managed persistence is to be accomplished with a user-defined persistence mechanism, the component developer must implement the various persistence classes defined in the persistence framework.

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

62.2.9.2 *Self-managed Persistence*

Self-managed persistence is also supported by the **entity** container API type. Like container-managed persistence, the component developer has two choices: to use the CORBA persistent state service or some user-defined persistence mechanism. But since no declarations are available to support code generation, the component developer is responsible for implementing both the callback interfaces and the persistence classes. The container supports access to a component persistence abstraction provided by the CORBA persistent state service, which hides many of the details of the underlying persistence mechanism from the component developer.

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

62.2.10 *Application Operation Invocation*

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

Application operations may raise exceptions, both application exceptions (i.e. those defined as part of the IDL interface definition) and system exceptions (those that are not). Exceptions defined as part of the IDL interfaces defined for a component (that includes both provided interfaces and supported interfaces) are raised back to the client directly and do not affect the current transaction. All other exceptions raised by the application are intercepted by the container which then raises the **TRANSACTION_ROLLEDBACK** exception to the client, if a transaction is active. Otherwise they are reported back to the client directly.

62.2.11 *Component Implementations*

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

62.2.12 *Component Levels*

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

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

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

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

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

- Basic container APIs are defined in Section 62.3.
- Extended container APIs are defined in Section 62.4.

Partitioning the component function into two discrete packages permits the EJB 1.1 APIs to be used to implement basic CORBA components in Java. It also supports the construction of CORBA components in any supported CORBA language which can be accessed by EJB clients. This is described further in Chapter 64, "Integrating with Enterprise JavaBeans".

62.2.13 Component Categories

As indicated in Section 62.1.4, this specification defines four component categories whose behavior is specified by the two **container API types**. Additionally we reserve a component category to describe the empty container (i.e. a container API type which does not use one of the API frameworks defined in this specification). The four component categories are described briefly in the following sections. The component categories are independent of the component levels defined in Section 62.2.12.

62.2.13.1 The Service Component

The **service** component is a CORBA component with the following properties:

- no state
- no identity
- behavior

The lifespan of a **service** component is equivalent to the lifetime of a single operation request (i.e. **method**) so it is useful for functions such as command objects which have no duration beyond the lifetime of a single client interaction with them. A service component can also be compared to a traditional TP monitor program like a Tuxedo service or a CICS transaction. A service component provides a simple way of wrapping existing procedural applications.

A service component is equivalent to a stateless EJB session bean.

The following table (Table 62-6) summarizes the characteristics of a service component as seen by the server programmer:

Table 62-6 Service Component Design Characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus SessionContext (basic) Session2Context (extended)
Callback Interfaces	SessionComponent
CORBA Usage Model	stateless
External API Types	keyless
Client Design Pattern	Factory
Persistence	No
Servant Lifetime Policy	method
Transactions	May use, but not included in current transaction
Events	Transactional or Non-transactional
Executor	Single segment with a single servant and no managed storage

Because of its absence of state, any programming language servant can service any **ObjectId**, enabling such servants to be managed as a pool or dynamically created as required, depending on usage patterns. Because a service component has no identity, **ObjectIds** can be managed by the POA, not the component implementor, and the client sees only the factory design pattern.

The service component can use either container-managed or self-managed transactions.

62.2.13.2 The Session Component

The **session** component is a CORBA component with the following properties:

- transient state
- identity which is not persistent
- behavior

The lifespan of a **session** component is specified using the servant lifetime policies defined in Section 62.2.5. A session component (with a **transaction** lifetime policy) is similar to an MTS component and is useful for modeling things like iterators, which require transient state for the lifetime of a client interaction but no persistent store. A session component is equivalent to the stateful session bean found in EJB.

The following table (Table 62-7) summarizes the characteristics of a session component as seen by the server programmer:

Table 62-7 Session Component Design Characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base Set plus SessionContext (basic) Session2Context (extended)
Callback Interfaces	SessionComponent plus (optionally) SessionSynchronization
CORBA usage model	conversational
Client Design Pattern	Factory
External API Types	keyless
Persistence	No
Servant Lifetime Policy	Any
Transactions	May use, but not included in current transaction
Events	Transactional or Non-transactional
Executor	Single segment with a single servant and no managed storage

A programming language servant is allocated to an **ObjectId** for the duration of the servant lifetime policy specified. At that point, the servant can be returned to a pool and re-used for a different **ObjectId**. Alternatively, servants may be dynamically created as required, depending on usage patterns. Because a session component has no persistent identity, **ObjectIds** can be managed by the container, however extended components may choose to participate in creating references if desired, and the client sees only the factory design pattern.

The session component shall use either container-managed or self-managed transactions.

62.2.13.3 The Process Component

The process component is a CORBA component with the following properties:

- persistent state which is not visible to the client and is managed by the **process** component implementation or the container
- persistent identity which is managed by the **process** component and can be made visible to the client only through user-defined operations
- behavior which may be transactional.

The process component is intended to model objects that represent business processes (e.g. applying for a loan, creating an order, etc.) rather than entities (e.g. customers, accounts, etc.). The major difference between **process** components and **entity** components is that the **process** component does not expose its persistent identity to the client (except through user-defined operations).

The following table (Table 62-8) summarizes the characteristics of process component as seen by the server programmer:

Table 62-8 Process Component Design Characteristics

Design Characteristic	Property
External Interfaces	As defined in component IDL
Internal Interfaces	Base set plus EntityContext (basic) Entity2Context (extended)
Callback Interfaces	EntityComponent
CORBA usage model	durable
Client Design Pattern	Factory
External API Types	keyless
Persistence	Self-managed with or without PSS or Container-managed with or without PSS
Servant Lifetime Policy	Any
Transactions	May use, and can be included in current transaction
Events	Non-transactional or transactional events
Executor	Multiple segments with associated managed storage

A process component may have transactional behavior. The container will interact with the CORBA transaction service to participate in the commit process. The process component shall use container-managed transactions. This is identical to the EJB restriction for Entity Beans.

The process component can use **container-managed** or **self-managed** persistence using either the CORBA persistent state service or a user-defined persistence mechanism. The implications of the various choices are described in Section 62.2.9. The entity container uses callback interfaces which enable the process component's implementation to retrieve and save state data at activation and passivation respectively.

62.2.13.4 The Entity Component

The **entity** component is a CORBA component with the following properties:

- persistent state which is visible to the client and is managed by the **entity** component implementation or the container
- identity which is architecturally visible to its clients through a **primaryKey** declaration
- behavior which may be transactional.

As a fundamental part of the architecture, **entity** components expose their persistent state to the client as a result of declaring a **primaryKey** value on their home declaration. The entity component may be used to implement the entity bean in EJB.

The following table (Table 62-9) summarizes the characteristics of **entity** component as seen by the server programmer:

Table 62-9 Entity Component Design Characteristics

Design Characteristic	Property
External Interfaces	As defined in the component IDL
Internal Interfaces	Base set plus EntityContext (basic) Entity2Context (extended)
Callback Interfaces	EntityComponent
CORBA usage model	durable
Client Design Pattern	Factory or Finder
External API Types	keyfull
Persistence	Self-managed with or without PSS or Container-managed with or without PSS
Servant Lifetime Policy	Any
Transactions	May use, and can be included in current transaction
Events	Non-transactional or transactional events
Executor	Multiple segments with associated managed storage

The entity component shall use container-managed transactions. The container shall interact with the CORBA transaction service to participate in the commit process. This is identical to the EJB restriction for Entity Beans.

The entity component can use **container-managed** or **self-managed** persistence using either the CORBA persistent state service or a user-defined persistence mechanism. The implications of the various choices are described in Section 62.2.9. The entity container uses callback interfaces which enable the entity component's implementation to retrieve and save state data at activation and passivation, respectively.

62.3 Server Programming Interfaces - Basic Components

This section defines the local interfaces used and provided by the component developer for basic components. These interfaces are then grouped as follows:

- interfaces common to both container API types
- interfaces supported by the session container API type only
- interfaces supported by the entity container API type only

Unless otherwise indicated, all of these interfaces are defined within the **Basic** module embedded within the **Components** module.

62.3.1 Component Interfaces

All components deal with three sets of interfaces:

- **internal** interfaces which are used by the component developer and provided by the container to assist in the implementation of the component's behavior,
- **external** interfaces which are used by the client and implemented by the component developer, and
- **callback** interfaces which are used by the container and implemented by the component, either in generated code or directly, in order for the component to be deployed in the container.

A container API type defines a base set of internal interfaces which the component developers use in their implementation. These interfaces are then augmented by others that are unique to the component category being developed.

- **CCMContext** - which serves as a bootstrap and provides accessors to the other internal interfaces including access to the runtime services implemented by the container.

Each container API type has its own specialization of **CCMContext** which we refer to as a context.

- **UserTransaction** - which wraps the demarcation subset of the CORBA transaction service required by the application developer.
- **EnterpriseComponent** - which is the base class that all **callback** interfaces derive from.

All components implement a callback interface which is determined by the component category. It serves the same role as **EnterpriseBean** in EJB.

When a component instance is instantiated in a container, it is passed a reference to its context, a local interface used to invoke services. For basic components, these services include transactions and security. The component uses this reference to invoke operations required by the implementation at runtime beyond what is specified in its deployment descriptor.

62.3.2 Interfaces Common to both Container API Types

This section describes the interfaces and operations provided by both **container API types** to support all categories of CORBA components.

62.3.2.1 The CCMContext Interface

The **CCMContext** is an **internal** interface which provides a component instance with access to the common container-provided runtime services applicable to both **container API types**. It serves as a “bootstrap” to the various services the container provides for the component.

The **CCMContext** provides the component access to the various services provided by the container. It enables the component to simply obtain all the references it may require to implement its behavior.

```
typedef SecurityLevel2::Credentials Principal; exception IllegalState { };
```

```
local interface CCMContext {
    Principal get_caller_principal();
    CCMHome get_CCM_home();
    boolean get_rollback_only() raises (IllegalState);
    Transaction::UserTransaction get_user_transaction()
        raises (IllegalState);
    boolean is_caller_in_role (in string role);
    void set_rollback_only() raises (IllegalState);
};
```

get_caller_principal

The **get_caller_principal** operation obtains the CORBA security credentials in effect for the caller. Security on the server is primarily controlled by the security policy in the deployment descriptor for this component. The component may use this operation to determine the credentials associated with the current client invocation.

get_CCM_home

The **get_CCM_home** operation is used to obtain a reference to the home interface. The home is the interface which supports factory and finder operations for the component and is defined by the **home** declaration in component IDL.

get_rollback_only

The **get_rollback_only** operation is used by a component to test if the current transaction has been marked for rollback. The **get_rollback_only** operation returns **TRUE** if the transaction has been marked for rollback, otherwise it returns **FALSE**. If no transaction is active, the **IllegalState** exception shall be raised. When **get_rollback_only** is issued by a component, it results in a **CosTransaction::Current::get_status** being issued to the CORBA transaction service and the **status** value returned being tested for the **MARKED_ROLLBACK** state.

get_user_transaction

The **get_user_transaction** operation is used to access the **Transaction::UserTransaction** interface. The **UserTransaction** interface is used to implement self-managed transactions. The **IllegalState** exception shall be raised if this component is using container-managed transactions.

is_caller_in_role

The **is_caller_in_role** operation is used by the CORBA component to compare the current credentials to the credentials defined by the role parameter. If they match, **TRUE** is returned. If not, **FALSE** is returned.

set_rollback_only

The **set_rollback_only** operation is used by a component to mark an existing transaction for abnormal termination. If no transaction is active, the **IllegalState** exception shall be raised. When **set_rollback_only** is issued by a component, it results in a **CosTransaction::Current::rollback_only** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation.

62.3.2.2 *The Home Interface*

A home is an **external** interface which supports factory and finder operations for the component. These operations are generated from the **home** IDL declaration (see Section 61.7, “Homes,” on page 61-53). The context supports an operation (**get_CCM_home**) to obtain a reference to the component’s home interface.

62.3.2.3 *The UserTransaction Interface*

A CORBA component may use either container-managed or self-managed transactions, depending on the component category. With container-managed transactions, the component implementation relies on the transaction policy declarations packaged with the deployment descriptor and contains no transaction APIs in its implementation code.

This is identical to container-managed transactions in EJB or the default processing of an MTS component.

A component specifying self-managed transactions may use the CORBA transaction service directly to manipulate the current transaction or it may choose to use a simpler API, defined by this specification, which exposes only those transaction demarcation functions needed by the component implementation.

Manipulation of the current transaction shall be consistent between the client, the transaction policy specified in the deployment descriptor, and the component implementation.

*For example, if the client or the container starts a transaction, the component may not end it (**commit** or **rollback**). The rules to be used are defined by the CORBA transaction service.*

If the component uses the **CosTransactions::Current** interface, all operations defined for **Current** may be used as defined by the CORBA transaction service with the following exceptions:

- The **Control** object returned by **suspend** may only be used with **resume**.
- Operations on **Control** are not supported with CORBA components and may raise the **NO_IMPLEMENT** system exception.

*The **Control** interface in the CORBA transaction service supports accessors to the **Coordinator** and **Terminator** interfaces. The **Coordinator** is used to build object versions of XA resource managers. The **Terminator** is used to allow a transaction to be ended by someone other than the originator. Since neither function is within the scope of the demarcation subset of CORBA transactions used with CORBA components, we allow CORBA transaction services implementations used with CORBA components to raise the **NO_IMPLEMENT** exception. This provides the same level of function as the **bean-managed** transaction policy in Enterprise JavaBeans.*

The **UserTransaction** is an **internal** interface implemented by the container and is defined within its own module, **Transaction**, within the **Components** module (**Components::Transaction**). Because the **UserTransaction** is a wrapper over **CosTransactions::Current**, it is thread specific. The **UserTransaction** exposes a simple demarcation subset of the CORBA transaction service to the component. The context supports an operation (**get_user_transaction**) to obtain a reference to the **UserTransaction** interface. The **UserTransaction** interface is defined by the following IDL:

```

typedef sequence<octet> TranToken;
exception NoTransaction { };
exception NotSupported { };
exception SystemError { };
exception Rollback { };
exception HeuristicMixed { };
exception HeuristicRollback { };
exception Security { };
exception InvalidToken { };

enum Status {
    ACTIVE,
    MARKED_ROLLBACK,
    PREPARED,
    COMMITTED,
    ROLLED_BACK,
    NO_TRANSACTION,
    PREPARING,
    COMMITTING,
    ROLLING_BACK
};

local interface Transaction {
    void begin () raises (NotSupported, SystemError);
    void commit () raises (Rollback, NoTransaction,
                          HeuristicMixed, HeuristicRollback,
                          Security, SystemError);
    void rollback () raises (NoTransaction, Security, SystemError);
    void set_rollback_only () raises (NoTransaction, SystemError);
    Status get_status() raises (SystemError);
    void set_timeout (in long to) raises (SystemError);
    TranToken suspend () raises (NoTransaction, SystemError);
    void resume (in TranToken txtoken)
                raises (InvalidToken, SystemError);
};

```

begin

The **begin** operation is used by a component to start a new transaction and associate it with the current thread. When **begin** is issued by a component, it results in a **CosTransaction::Current::begin** with **report_heuristics** set to **TRUE** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation. The **NotSupported** exception is return if it is received from the CORBA transaction service. Since nested transactions are not supported by CORBA component containers, this indicates an attempt to start a new transaction when an existing transaction is active. All other exceptions are converted to the **SystemError** exception.

commit

The **commit** operation is used by a component to terminate an existing transaction normally. When **commit** is issued by a component, it results in a **CosTransaction::Current::commit** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation. If no transaction is active, the **NoTransaction** exception shall be raised. If the **TRANSACTION_ROLLEDBACK** system exception is returned, it is converted to the **Rollback** exception. The **CosTransaction::HeuristicMixed** and **CosTransaction::HeuristicRollback** exceptions are reported as the **HeuristicMixed** and **HeuristicRollback** exceptions respectively. The **NO_PERMISSION** system exception is converted to the **Security** exception. All other exceptions are converted to the **SystemError** exception.

rollback

The **rollback** operation is used by a component to terminate an existing transaction abnormally. When **rollback** is issued by a component, it results in a **CosTransaction::Current::rollback** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation. If no transaction is active, the **NoTransaction** exception shall be raised. The **NO_PERMISSION** system exception is converted to the **Security** exception. All other exceptions are converted to the **SystemError** exception.

set_rollback_only

The **set_rollback_only** operation is used by a component to mark an existing transaction for abnormal termination. When **set_rollback_only** is issued by a component, it results in a **CosTransaction::Current::rollback_only** being issued to the CORBA transaction service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation. If no transaction is active, the **NoTransaction** exception shall be raised. All other exceptions shall be converted to the **SystemError** exception.

get_status

The **get_status** operation is used by a component to determine the status of the current transaction. If no transaction is active, it returns the **NoTransaction** status value. Otherwise it returns the state of the current transaction. When **get_status** is issued by a component, it results in a **CosTransaction::Current::get_status** being issued to the CORBA transaction service. The status values returned by this operation are equivalent to the status values of its corresponding CORBA transaction service operation. All exceptions shall be converted to the **SystemError** exception.

set_timeout

The **set_timeout** operation is used by a component to associate a time-out value with the current transaction. The timeout value (**to**) is specified in seconds. When **set_timeout** is issued by a component, it results in a **CosTransaction::Current::set_timeout** being issued to the CORBA transaction

service. The rules for the use of this operation are equivalent to the rules of its corresponding CORBA transaction service operation. All exceptions are converted to the **SystemError** exception.

suspend

The **suspend** operation is used by a component to disconnect an existing transaction from the current thread. The **suspend** operation returns a **TranToken** which can only be used in a subsequent **resume** operation. When **suspend** is issued by a component, it results in a **CosTransaction::Current::suspend** being issued to the CORBA transaction service. The rules for the use of this operation are more restrictive than the rules of its corresponding CORBA transaction service operation:

- Only one transaction may be suspended
- The suspended transaction is the only transaction that may be resumed.

If no transaction is active, the **NoTransaction** exception shall be raised. All other exceptions are converted to the **SystemError** exception.

resume

The **resume** operation is used by a component to reconnect a transaction previously suspended to the current thread. The **TranToken** identifies the suspended transaction which is to be resumed. If the transaction identified by **TranToken** has not been suspended, the **InvalidToken** exception shall be raised. When **resume** is issued by a component, it results in a **CosTransaction::Current::resume** being issued to the CORBA transaction service. The rules for the use of this operation are more restrictive than the rules of its corresponding CORBA transaction service operation since the single suspended transaction is the only transaction that may be resumed. All other exceptions are converted to the **SystemError** exception.

*The **UserTransaction** interface is equivalent to the **UserTransaction** interface (`javax.transaction.UserTransaction`) in EJB with the addition of the **suspend** and **resume** operations.*

62.3.2.4 The EnterpriseComponent Interface

All CORBA components must implement an interface derived from the **EnterpriseComponent** interface to be housed in a component container. **EnterpriseComponent** is a **callback** interface which defines no operations.

```
local interface EnterpriseComponent { };
```

62.3.3 Interfaces Supported by the Session Container API Type

This section describes the interfaces supported by the session container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this container API type.

62.3.3.1 *The SessionContext Interface*

The **SessionContext** is an **internal** interface which provides a component instance with access to the container-provided runtime services. It serves as a “bootstrap” to the various services the container provides for the component.

The **SessionContext** enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };
```

```
local interface SessionContext : CCMContext {
    Object get_CCM_object() raises (IllegalState);
};
```

get_CCM_object

The **get_CCM_object** operation is used to get the reference used to invoke the component. For basic components, this will always be the component reference. For extended components, this will be a specific facet reference. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

62.3.3.2 *The SessionComponent Interface*

The **SessionComponent** is a **callback** interface implemented by a session CORBA component. It provides operations for disassociating a context with the component and to manage servant lifetimes for a session component.

```
enum CCMExceptionReason {
    SYSTEM_ERROR,
    CREATE_ERROR,
    REMOVE_ERROR,
    DUPLICATE_KEY,
    FIND_ERROR,
    OBJECT_NOT_FOUND,
    NO_SUCH_ENTITY};
```

```
exception CCMException {CCMExceptionReason reason};
```

```
local interface SessionComponent : EnterpriseComponent {
    void set_session_context ( in SessionContext ctx)
        raises (CCMException);
    void ccm_activate() raises (CCMException);
    void ccm_passivate() raises (CCMException);
    void ccm_remove () raises (CCMException);
};
```


set_session_context

The **set_session_context** operation is used to set the **SessionContext** of the component. The container calls this operation after a component instance has been created. This operation is called outside the scope of an active transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_activate

The **ccm_activate** operation is called by the container to notify a session component that it has been made active. The component instance should perform any initialization required prior to operation invocation. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_passivate

The **ccm_passivate** operation is called by the container to notify a session component that it has been made inactive. The component instance should release any resources it acquired at activation time. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_remove

The **ccm_remove** operation is called by the container when the servant is about to be destroyed. It informs the component that it is about to be destroyed. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

62.3.3.3 *The SessionSynchronization Interface*

The **SessionSynchronization** interface is a **callback** interface which may optionally be implemented by the session component. It permits the component to be notified of transaction boundaries by its container.

```
exception CCMException {CCMExceptionReason reason};
```

```
local interface SessionSynchronization {
    void after_begin () raises (CCMException);
    void before_completion () raises (CCMException);
    void after_completion (
        in boolean committed) raises (CCMException);
};
```

after_begin

The **after_begin** operation is called by the container to notify a session component that a new transaction has started, and that the subsequent operations will be invoked in the context of the transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

before_completion

The **before_completion** operation is called by the container just prior to the start of the two-phase commit protocol. The container implements the **CosTransactions::Synchronization** interface of the CORBA transaction service and invokes the **before_completion** operation on the component before starting its own processing. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

after_completion

The **after_completion** operation is called by the container after the completion of the two-phase commit protocol. If the transaction has committed the **committed** value is set to **TRUE**. If the transaction has been rolled back, the **committed** value is set to **FALSE**. The container implements the **CosTransactions::Synchronization** interface of the CORBA transaction service and invokes the **after_completion** operation on the component after completing its own processing. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

62.3.4 Interfaces Supported by the Entity Container API Type

This section describes the interfaces supported by the entity container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this container API type.

62.3.4.1 The EntityContext Interface

The **EntityContext** is an **internal** interface which provides a component instance with access to the container-provided runtime services. It serves as a “bootstrap” to the various services the container provides for the component.

The **EntityContext** enables the component to simply obtain all the references it may require to implement its behavior.

```
exception IllegalState { };
```

```
local interface EntityContext : CCMContext {
    Object get_CCM_object () raises (IllegalState);
    PrimaryKeyBase get_primary_key () raises (IllegalState);
};
```

get_CCM_object

The **get_CCM_object** operation is used to obtain the reference used to invoke the component. For basic components, this will always be the component reference. For extended components, this will be a specific facet reference. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

get_primary_key

The **get_primary_key** operation is used by an **entity** component to access the primary key value declared for this component's home. This operation is equivalent to issuing the same operation on the component's home interface. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

62.3.4.2 *The EntityComponent Interface*

The **EntityComponent** is a **callback** interface implemented by both process and entity components. It contains operations to manage the persistent state of the component.

Issue – As currently defined, any operation request will cause the container to activate the component segment, if required. Since the component reference is well-structured, we could consider the possibility of trapping navigation operations prior to activation and executing them without actually activating the component (or we could leave that to clever implementations).

```
exception CCMException {CCMExceptionReason reason};
```

```
local interface EntityComponent : EnterpriseComponent {
    void set_entity_context (in EntityContext ctx)
        raises (CCMException);
    void unset_entity_context ()raises (CCMException);
    void ccm_activate () raises (CCMException);
    void ccm_load ()raises (CCMException);
    void ccm_store ()raises (CCMException);
    void ccm_passivate ()raises (CCMException);
    void ccm_remove ()raises (CCMException);
};
```

set_entity_context

The **set_entity_context** operation is used to set the **EntityContext** of the component. The container calls this operation after a component instance has been created. This operation is called outside the scope of an active transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

unset_entity_context

The **unset_entity_context** operation is used to remove the **EntityContext** of the component. The container calls this operation just before a component instance is destroyed. This operation is called outside the scope of an active transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_activate

The **ccm_activate** operation is called by the container to notify the component that it has been made active. For most CORBA component implementations, no action is required. The component instance should perform any initialization (other than establishing its state) required prior to operation invocation. This operation is called within an unspecified transaction context. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_load

The **ccm_load** operation is called by the container to instruct the component to synchronize its state by loading it from its underlying persistent store. When container-managed persistence is implemented using the CORBA persistent state service, this operation can be implemented in generated code. If self-managed persistence is being used, the component is responsible for locating its state in a persistent store. This operation executes within the scope of the current transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_store

The **ccm_store** operation is called by the container to instruct the component to synchronize its state by saving it in its underlying persistent store. When container-managed persistence is implemented using the CORBA persistent state service, this operation can be implemented in generated code. If self-managed persistence is being used, the component is responsible for saving its state in the persistent store. This operation executes within the scope of the current transaction. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_passivate

The **ccm_passivate** operation is called by the container to notify the component that it has been made inactive. For most CORBA component implementations, no action is required. The component instance should perform any termination processing (other than saving its state) required prior to being passivated. This operation is called within an unspecified transaction context. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

ccm_remove

The **ccm_remove** operation is called by the container when the servant is about to be destroyed. It informs the component that it is about to be destroyed. This operation is always called outside the scope of a transaction. The component raises the **CCMException** with the **REMOVE_ERROR** minor code if it does not allow the destruction of the component. The component may raise the **CCMException** with the **SYSTEM_ERROR** minor code to indicate a failure caused by a system level error.

*The **EntityComponent** interface is equivalent to the **EntityBean** interface in Enterprise JavaBeans. Container-managed persistence with the CORBA persistent state service supports automatic code generation for **ccm_load** and **ccm_store**. For self-managed persistence, the component implementor provides the **ccm_load** and **ccm_store** methods. Since both process and entity components have persistent state and container-managed persistence, the same callback interfaces can be used.*

62.4 Server Programming Interfaces - Extended Components

This section defines the local interfaces used and provided by the component developer for extended components. These interfaces are grouped as in Section 62.3. Unless otherwise indicated, all of these interfaces are defined within the **Extended** module embedded within the **Components** module. Extended components add interfaces in the following areas:

- **CCM2Context** - adds functions unique to extended components.

Each container API type has its own specialization of **CCM2Context** which we refer to as a context. The context for extended components adds accessors to persistence and event services and supports operations for managing servant lifetime policy, and creating and managing object references in conjunction with the POA.

- **ComponentId** - encapsulates a component identifier, which is an abstract information model used to locate the component's state.

Only the **entity container API type** supports the **ComponentId** interface.

- **Event** - offers the subset of the CORBA notification service supported by CORBA components.

62.4.1 Interfaces Common to both Container API Types

This section describes the interfaces and operations provided for extended components by both **container API types** to support all categories of CORBA components.

62.4.1.1 The CCM2Context Interface

The **CCM2Context** is an **internal** interface which extends the **CCMContext** interface to provide the extended component instance with access to additional container-provided runtime services applicable to both **container API types**. These services include advanced persistence using the CORBA persistent state service, events using CORBA notification, and runtime management of component references and servants using the POA. The **CCM2Context** is defined by the following IDL:

```
typedef CosPersistentState::CatalogBase CatalogBase;
typedef CosPersistentState::Typeld Typeld;

exception PolicyMismatch { };
exception PersistenceNotAvailable { };

local interface CCM2Context : CCMContext {
    HomeRegistration get_home_registration ();
    Events::Event get_event();
    void req_passivate () raises (PolicyMismatch);
    CatalogBase get_persistence (in Typeld catalog_type_id)
        raises (PersistenceNotAvailable);
};
```

get_home_registration

The **get_home_registration** operation is used to obtain a reference to the **HomeRegistration** interface. The **HomeRegistration** is used to register component homes so they may be located by the **HomeFinder**.

get_event

The **get_event** operation is used to obtain a reference to the **Event** interface. The **Event** interface is used by the component to emit or publish events for external consumption or to subscribe to events it needs to process.

req_passivate

The **req_passivate** operation is used by the component to inform the container that it wishes to be passivated when its current operation completes. To be valid, the component must have a servant lifetime policy of **component** or **container**. If not the **PolicyMismatch** exception shall be raised.

get_persistence

The **get_persistence** operation provides the component access to a persistence framework provided by an implementation of the CORBA persistence state service. It returns a **CosPersistentState::CatalogBase** which serves as an index to the available storage homes. The **CatalogBase** is identified by its **CosPersistentState::TypeId catalog_type_id**. If the **CatalogBase** identified by **catalog_type_id** is not available on this container, the **PersistenceNotAvailable** exception shall be raised.

62.4.1.2 The HomeRegistration Interface

The **HomeRegistration** is an **internal** interface which may be used by the CORBA component to register its home so it can be located by a **HomeFinder**.

*The **HomeRegistration** interface allows a component implementation to advertise a home instance that can be used to satisfy a client's find_home request. It may also be used by an administrator to do the same thing. It is likely that the combination of **HomeRegistration** and **HomeFinder** interfaces will work within the domain of a single container provider unless multiple implementations use other shareable directory mechanisms, e.g. an LDAP global directory. Federating **HomeFinders** is a similar problem to federating CORBA security domains and we defer to the security people for an architecture for such federation rather than attempting to specify such an architecture in this specification.*

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

```
local interface HomeRegistration {
    void register_home (
        in CCMHome home,
        in string home_name);
    void unregister_home (in CCMHome home);
};
```

register_home

The **register_home** operation is used to register a component home with the **HomeFinder** so it can be located by a component client. The **home** parameter identifies the home being registered and can be used to obtain both the **IR::ComponentDef (CCMHome::get_component_def)** and the **IR::InterfaceDef (CORBA::Object::get_interface_def)** to support both **HomeFinder::find_home_by_component_type** and **HomeFinder::find_home_by_home_type**. The **home_name** parameter identifies an Interoperable Naming Service (INS) name that can be used as input to the **HomeFinder::find_home_by_name** operation. If the **home_name** parameter is NULL, no name is associated with this home so this home cannot be retrieved by name.

unregister_home

The **unregister_home** operation is used to remove a component home from the **HomeFinder**. Once **unregister_home** completes, a client will never be returned a reference to the home specified as being unregistered. The **home** parameter identifies the home being unregistered.

62.4.1.3 The ProxyHomeRegistration Interface

Because CORBA components exploit the dynamic activation features of the POA, it is possible for some component types to provide a home which is not collocated with the component instances it creates. This permits load balancing criteria to be applied in selecting the actual server and POA where this instance will be created. The **ProxyHomeRegistration** is an **internal** interface, derived from **HomeRegistration**, which can be used by the CORBA component to register a remote home (i.e. one that is **NOT** collocated with the component) so it can be returned by a **HomeFinder**. The **ProxyHomeRegistration** interface is defined by the following IDL:

```
exception UnknownActualHome { };
exception ProxyHomeNotSupported { };

local interface ProxyHomeRegistration : HomeRegistration {
    void register_proxy_home (
        in CCMHome rhome,
        in CCMHome ahome)
        raises (UnknownActualHome, ProxyHomeNotSupported);
};
```

register_proxy_home

The **register_proxy_home** operation is used to register a component home, not collocated with the instances that it can create, with the **HomeFinder** so the proxy home can be used by component clients. The **rhome** parameter identifies the proxy home being registered. The **ahome** parameter identifies the actual home which the **rhome** is associated with. If the actual home specified by **ahome** is not known, the **UnknownActualHome** exception shall be raised. If this component does not support proxy homes, the **ProxyHomeNotSupported** exception shall be raised. Support for proxy homes is a component implementation option.

62.4.1.4 The Event Interface

The **Event** is an **internal** interface which supports operations for emitting and publishing events and for subscribing to events emitted or published by others. The **Event** and **LocalCookie** interfaces are defined in their own module (**Components::Events**) and provide a simple mechanism for connecting the component to a CORBA notification channel established and managed by the container. The implementations of the operations generated from the **emits**, **publishes**, and **consumes** declaration in the component's IDL (see Section 61.6, "Events," on

page 61-44) delegate to these interfaces. The context supports an operation (**get_event**) to obtain a reference to the **Event** interface. The **Event** interface is defined by the following IDL:

```

typedef CosNotification::EventHeader EventHeader;
typedef CosNotifyChannelAdmin::ChannelId Channel;

exception ChannelUnavailable { };
exception InvalidSubscription { };
exception InvalidName { };
exception InvalidChannel { };

local interface LocalCookie {
    boolean same_as (in LocalCookie cookie);};
local interface Event {
    EventConsumerBase create_channel
    (out Channel chid)
    raises (ChannelUnavailable);
LocalCookie subscribe (
    in EventConsumerBase ecb,
    in Channel chid)raises (ChannelUnavailable);
void unsubscribe (in LocalCookie cookie)
    raises (InvalidSubscription);
EventConsumerBase obtain_channel (
    in string supp_name,
    in EventHeader hdr) raises (InvalidName);
void listen (in EventConsumerBase ecb,
    in string csmr_name) raises (InvalidName);
void push (in EventBase evt);
void destroy_channel (in Channel chid)raises (InvalidChannel);
};

```

same_as

The **same_as** operation compares two **LocalCookie** instances for equivalence and returns **TRUE** if equivalent, otherwise it returns **FALSE**.

create_channel

The **create_channel** operation is used by a component to bind a notification channel to be used to push component events. This operation corresponds to a **publishes** declaration in component IDL. It returns an **EventConsumerBase** which can be used to push events into the channel. When a **create_channel** operation is issued by a component, the container interacts with CORBA notification to create an event channel for the components exclusive use. If the container cannot connect to the channel, the **ChannelUnavailable** exception shall be raised. The **chid** is returned to the component as an identifier of the channel.

subscribe

The **subscribe** operation allows the component to express interest in receiving one or more events. The **ecb** identifies an **EventConsumerBase** which the container will use to push the event to the component. If the container is not connected to the channel, the **ChannelUnavailable** exception shall be raised. The **EventConsumerBase** must implement the **push** operation defined by the **<event_type>Consumer** interface. The **subscribe** operation returns a **cookie** which is used to delete the subscription.

unsubscribe

The **unsubscribe** operation deletes the subscription specified by the **cookie** previously returned by **subscribe**. If no subscription is associated with the **cookie**, the **InvalidSubscription** exception shall be raised.

obtain_channel

The **obtain_channel** operation is used by the component to obtain an **EventConsumerBase** which it can use to push events. This operation corresponds to an **emits** declaration in component IDL. The **supp_name** string identifies an Interoperable Naming Service (INS) name which is used to identify the **SupplierAdmin** to be used by CORBA notification. The name is associated with the **SupplierAdmin** through container specific configuration data. The **obtain_channel** operation may optionally specify the **EventHeader** required by CORBA notification which will be used for all events pushed to this channel. If **hdr** is present, it is prefixed to all events pushed to this channel. If not, it is defaulted as described in Section 66.4, “Event Management Integration,” on page 66-252. If the **supp_name** is not recognized, the **InvalidName** exception shall be raised.

listen

The **listen** operation is used by the component to inform the container that it would like to receive events of a particular type. This corresponds to the **consumes** declaration in component IDL. The **cmsr_name** string identifies an INS name which is used to identify the **ConsumerAdmin** to be used by CORBA notification. The name is associated with the **ConsumerAdmin** through container specific configuration data. The component provides an **EventConsumerBase** interface that implements the **push** operation on the **<event_type>Consumer** interface. If the **cmsr_name** is not recognized, the **InvalidName** exception shall be raised.

push

The **push** operation is used by a component to transmit an event. The event **evt** is a valuetype derived from **EventBase**.

destroy_channel

The **destroy_channel** operation is used by a component to delete the channel identified by **chid**. The **InvalidChannel** exception can be raised if the **chid** parameter is not the value previously returned by **create_channel**.

EJB does not have an event API yet, but one is under development. The Java 2 Platform, Enterprise Edition (J2EE) does however have a messaging API (JMS) which supports publish/subscribe. This is an area that will need to be harmonized with EJB in the future.

62.4.2 Interfaces Supported by the Session Container API Type

This section describes the interfaces supported for extended components by the session container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this container API type.

62.4.2.1 The Session2Context Interface

The **Session2Context** is an **internal** interface which extends the **SessionContext** to provides a component instance with access to additional container-provided runtime services for the session container API type. It adds the ability to create references for components deployed in a **session** container API type. The **Session2Context** is defined by the following IDL:

```
enum BadComponentReferenceReason {
    NON_LOCAL_REFERENCE,
    NON_COMPONENT_REFERENCE,
    WRONG_CONTAINER,
};

exception BadComponentReference {
    BadComponentReferenceReason reason
};
exception IllegalState { };

local interface Session2Context : SessionContext, CCM2Context {
    Object create_ref (in CORBA::RepositoryId repid);
    Object create_ref_from_oid (
        in PortableServer::ObjectId oid,
        in CORBA::RepositoryId repid);
    PortableServer::ObjectId get_oid_from_ref (in Object ref)
        raises (IllegalState, BadComponentReference);
};
```

create_ref

The **create_ref** operation is used to create a reference to be exported to clients to invoke operations. The **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

create_ref_from_oid

The **create_ref_from_oid** operation is used to create a reference to be exported to clients which includes information provided by the component which it can use on subsequent operation requests. The **oid** parameter identifies the **ObjectId** to be encapsulated in the reference and the **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

get_oid_from_ref

The **get_oid_from_ref** operation is used by the component to extract the **oid** encapsulated in the reference. The **ref** parameter specifies the reference which contains the **oid**. This operation must be called within an operation invocation. If not the **IllegalState** exception shall be raised. If the reference was not created by this container, the **BadComponentReference** with the **WRONG_CONTAINER** minor code is raised.

62.4.3 Interfaces Supported by the Entity Container API Type

This section describes the interfaces provided for extended components by the entity container API type. This includes both **internal** interfaces provided by the container and **callback** interfaces which must be implemented by components deployed in this container API type.

62.4.3.1 Component Identifiers

The **ComponentId** interface is an **internal** interface provided by the entity container API type through which the component implementation and the container exchange identity information, referred to as *component identifiers*. The **ComponentId** interface encapsulates a component identifier, which is an abstract information model. The **ComponentId** interface is used in the following ways:

- Component implementations (usually home executor implementations) create component identifiers to describe new components, and to create object references that encapsulate the provided description. The **Entity2Context** interface acts as a factory for component identifiers and as the factory for object references.
- The container encodes the information encapsulated by the component identifier in the object identifier value it uses internally to create the object reference on the encapsulated POA. The encoding is not specified, since a container's choice of encoding does not affect interoperability or portability.
- While dispatching an incoming request, the container extracts and decodes the component identifier from the **ObjectId**. The extracted component identifier is made available to the component executor through the context before the request is dispatched to the component.
- When the container invokes **ccm_load** in the component executor, the implementation of **ccm_load** uses the contents of the component identifier to locate and incarnate the required component state.

In the following discussions, component identifiers and component object references are sometimes used as though the terms were synonymous. Since there is a one-to-one relationship between a component identifier and an object reference created from the component identifier, this discussion occasionally uses the term “component reference” to mean “the component reference created from the component identifier in question”, for the sake of brevity.

The **ComponentId** interface does not explicitly specify the state representation it encapsulates. The abstract state is implied by the interface and reflects the structure of the executor it describes (see Chapter 615, “CCM Implementation Framework” for a complete discussion of executor structure).

A component identifier encapsulates the following information:

- A *facet identifier* value denoting the target facet of the component reference
- A *segment identifier* value denoting the target segment of the component reference (i.e., the segment that supports the target facet)
- A sequence of *segment descriptors*

A segment descriptor includes the following:

- A segment identifier denotes the segment being described
- A *state identifier* value that denotes the persistent state of the segment in some storage mechanism.

A monolithic executor is represented as a degenerate case of the generalized component identifier, where the target segment identifier is set to zero and the sequence of segment descriptors contains a single element, whose segment identifier is zero and whose state identifier denotes the persistent state of the component’s single segment.

The facet identifier value zero is reserved to denote the component facet, i.e., the facet that supports the component equivalent interface. The segment identifier value zero is reserved to denote the segment that supports the component facet. For monolithic executors, the segment identifier values is always zero.

State identifier is an abstraction that generalizes a variety of possible state identity schemes. This specification provides a mechanism for describing state identifiers that can be extended by component implementors, allowing customization for storage mechanisms that do not support the standard persistence interfaces.

The **ComponentId** local interface and supporting constructs are defined by the following IDL:

```

typedef short SegmentId;
const SegmentId COMPONENT_SEGMENT = 0;

typedef short FacetId;
const FacetId COMPONENT_FACET = 0;

typedef sequence<octet> IdData;
typedef CosPersistentState::pid PersistentId;

exception InvalidStateIdData {};

typedef short StateIdType;
const StateIdType PERSISTENT_ID = 0;

abstract valuetype StateIdValue {
    StateIdType get_sid_type();
    IdData get_sid_data();
};

local interface StateIdFactory {
    StateIdValue create (in IdData data) raises (InvalidStateIdData);
};

valuetype PersistentIdValue : StateIdValue {
    private PersistentId pid;
    PersistentId get_pid();
    init (in PersistentId pid);
};

valuetype SegmentDescr {
    private StateIdValue sid;
    private SegmentId seg;
    StateIdValue get_sid();
    SegmentId get_seg_id();
    init (in StateIdValue sid, in SegmentId seg);
};

typedef sequence<SegmentDescr> SegmentDescrSeq;

local interface ComponentId {
    FacetId get_target_facet();
    SegmentId get_target_segment();
    StateIdValue get_target_state_id (in StateIdFactory sid_factory)
        raises (InvalidStateIdData);
    StateIdValue get_segment_state_id (
        in SegmentId seg,
        in StateIdFactory sid_factory)
        raises (InvalidStateIdData);
    ComponentId create_with_new_target (
        in FacetId new_target_facet,
        in SegmentId new_target_segment);
};

```

```

        SegmentDescrSeq get_segment_descrs (
            in StatelIdFactory sid_factory)
            raises (InvalidStatelIdData);
    };

```

62.4.3.2 *StateIdValue abstract valuetype*

The **StatelIdValue** type is the base valuetype for concrete, storage-specific state identity values. The container interacts with state identities completely in terms of this interface. A single pre-defined concrete value type derived from **StatelIdValue** is provided for **PersistentId** state identities. Component implementors, or suppliers of storage mechanisms that do not support the CORBA component persistence model can provide their own state identity types by deriving from **StatelIdValue** and implementing the required behaviors properly.

get_sid_type

The **get_sid_type** operation returns a discriminator (physically, a short) that identifies the type of the state identity encapsulated by the **StatelIdValue**. This specification defines the value zero (0) to denote a **Components::Extended::PersistentId** state identifier.

Issue – do we need to define this as an OMG-allocated space?

get_sid_data

The **get_sid_data** operation returns the encapsulated state identity expressed in a canonical form, as a sequence of octets. The implementation of the derived concrete value type is responsible for converting its encapsulated data into this form, and for supplying a factory which can construct an instance of the concrete type from an **IdData** value (a sequence of octets).

62.4.3.3 *StateIdFactory Interface*

StatelIdFactory is the abstract base interface for factories of state identity values derived from **StatelIdValue**. An implementation of **StatelIdFactory** must be supplied with the implementation of a concrete state identity type. If the **IdData** octet sequence provided in the **data** parameter cannot be decoded to create a proper instance of the expected state identity concrete type, the operation raises an **InvalidStatelIdData** exception.

create

The **create** operation constructs an instance of a concrete state identifier from the octet sequence parameter. This operation performs the inverse of the transformation performed by the **get_sid_data**.

62.4.3.4 *PersistentIdValue* valuetype

The **PersistentIdValue** type is a specialization of **StateIdValue** that encapsulates a **PersistentId** value for inclusion in a component identifier.

get_pid

The **get_pid** operation returns the **PersistentId** value encapsulated by the value type.

init

The initializer for **PersistentIdValue** creates an instance of the valuetype that encapsulates the **PersistentId** value passed as a parameter.

get_sid_value

The implementation of **get_sid_value** for **PersistentIdValue** performs no transformation on the encapsulated **PersistentId** value. The sequence of octets returned by **get_sid_value** is identical to the encapsulated **PersistentId** value.

62.4.3.5 *SegmentDescr* valuetype

The **SegmentDescr** type describes an executor segment, encapsulating a segment identifier and a state identifier. A component identifier for a segmented executor encapsulates a sequence of **SegmentDescr** instances.

get_sid

The **get_sid** operation returns the state identity value of the segment being described.

get_seg_id

The **get_seg_id** operation returns the segment identifier of the segment being described.

init

This initializer sets the value of the encapsulated segment identifier and state identifier to the values of the respective parameters.

62.4.3.6 *ComponentId* Interface

The **ComponentId** interface encapsulates a complete component identity. Instances of **ComponentId** can only be created by the **Entity2Context** interface, which is supplied by the container, or by duplicating an existing component identifier with a new target value, with **ComponentId::create_with_new_target**. Instances of **ComponentId** are also provided by the **EntityContext** interface in the context of a CORBA invocation. The value of the component identifier provided by the **Entity2Context** shall be identical to the component identifier value used to create the object reference on which the invocation was made. The **ComponentId** interface is a

read-only interface. Once a component identifier is constructed by the **create_component_id** operation or constructed internally and provided through the **Entity2Context** interface, the value of the component identifier cannot be altered.

get_target_facet

The **get_target_facet** operation returns the facet identifier of the facet which is the target of the component reference, i.e., the target of requests made on the component reference.

get_target_segment

The **get_target_segment** operation returns the segment identifier of the target segment, i.e., the segments that provides the target facet.

get_target_state_id

The **get_target_state_id** operation returns the state identifier of the target segment. The **StateIdFactory** specified in the **sid_factory** parameter is used by the implementation of **get_target_state_id** to construct the proper state identifier from the octet sequence encapsulated by the component identifier. If the state identifier of the target segment is a **PersistentIdValue**, the **sid_factory** parameter may be nil. Container implementations shall provide a default implementation of **StateIdFactory** to be used when the encapsulated state identifier value is a **PersistentIdValue**. If provided (or default) factory cannot construct a correct state identifier of the expected type from the undecoded octet sequence encapsulated by the component identifier, the operation raises an **InvalidStateIdData** exception.

get_segment_state_id

The **get_segment_state_id** operation returns the state identifier of the segment specified by the **seg** parameter. The semantics are otherwise identical to **get_target_state_id**, with respect the meaning and use of the **sid_factory** parameter.

get_segment_descrs

The **get_segment_descrs** operation returns a sequence containing all of the segment descriptors encapsulated by the component identifier. The sequence is a copy of the encapsulated sequence. The state identifier factory in the **sid_factory** parameter (or the default) is used by the implementation of **get_segment_descrs** to construct state identifiers of the appropriate concrete subtype of **StateIdValue**. If provided (or default) factory cannot construct a correct state identifier of the expected type from the undecoded octet sequence encapsulated by the component identifier, the operation raises an **InvalidStateIdData** exception.

create_with_new_target

The **create_with_new_target** operation creates a new component identifier that is identical to the target component identifier, except that the target facet and target segment values are replaced with the values of the **new_target_facet** and **new_target_segment** parameters, respectively.

This operation is intended primarily to be used in implementing navigation operations.

62.4.3.7 The Entity2Context Interface

The **Entity2Context** is an **internal** interface which extends the **EntityContext** interface to provide the extended component with access to additional container-provided runtime services for managing object references and advanced persistence. Object references for components deployed in a entity container API type can choose to use the CORBA persistent state service or some user defined persistence mechanism. The **ComponentId** interface (defined in Section 62.4.3.6) encapsulates this distinction when a reference is to be used. The **Entity2Context** is defined by the following IDL.

```
exception BadComponentReference {
    BadComponentReferenceReason reason };
exception IllegalState { };

local interface Entity2Context : EntityContext, CCM2Context {
    ComponentId get_component_id ()
        raises (IllegalState);
    ComponentId create_component_id (
        in FacetId target_facet,
        in SegmentId target_segment,
        in SegmentDescrSeq seq_descrs);
    ComponentId create_monolithic_component_id (
        in FacetId target_facet,
        in StateIdValue sid);
    Object create_ref_from_cid (
        in CORBA::RepositoryId repid,
        in ComponentId cid);
    ComponentId get_cid_from_ref (
        in Object ref) raises (BadComponentReference);
};
```

get_component_id

The **get_component_id** operation is used to obtain a reference to the **ComponentId** interface. The **ComponentId** interface encapsulates a persistence identifier which can be used to access the component's persistence state. If this operation is issued outside of the scope of a **callback** operation, the **IllegalState** exception is returned.

create_component_id

The **create_component_id** operation creates a component identifier value, initializing it with the values specified in the parameters. The **target_facet** parameter contains the facet identifier of the target facet, the **target_segment** parameter contains

the segment identifier of the target segment, and the **seq_descrs** parameter contains a sequence of segment descriptors describing all of the segments that constitute the component executor.

create_monolithic_component_id

The **create_monolithic_component_id** operation provides a simplified signature for creating a component identifier value for monolithic executors, which have a single segment. The **target_facet** parameter contains the facet identifier of the target facet, and the **sid** parameter contains the state identifier for the single executor segment. The target segment identifier encapsulated by the component identifier is set to zero, and the sequence of segment descriptors encapsulated by the component identifier has a single element, initialized with segment identifier value zero, and state identifier value specified by the **sid** parameter.

create_ref_from_cid

The **create_ref_from_cid** operation is used by a component factory to create an object reference which can be exported to clients. The **cid** parameter specifies the **ComponentId** value to be placed in the object reference and made available (using the **get_component_id** operation on the context) when the **EntityComponent callback** operations are invoked. The **repid** parameter identifies the **RepositoryId** associated with the interface for which a reference is being created.

get_cid_from_ref

The **get_cid_from_ref** operation is used by a persistent component to retrieve the **ComponentId** encapsulated in the reference (**ref**). The **ComponentId** interface supports operations to locate the state in some persistent store. The **BadComponentReference** exception can be raised if the input reference is not local (**NON_LOCAL_REFERENCE**), not a component reference (**NON_COMPONENT_REFERENCE**), or created by some other container (**WRONG_CONTAINER**).

*The **ComponentId** structure is dependent on the home implementation and the container, in particular, its implementation of the **Entity2Context** interface. It is likely that a **ComponentId** created by one container will not be understandable by another, hence the possibility of the **WRONG_CONTAINER** exception.*

62.5 The Client Programming Model

This section describes the architecture of the component programming model as seen by the client programmer. The client programming model as defined by IDL extensions has been described previously (Chapter 61, “Component Model”). This section focuses on the use of standard CORBA by the client who wishes to communicate with a CORBA component implemented in a **Component Server**. It enables a CORBA client, which is not itself a CORBA component to communicate with a CORBA component.

The client interacts with a CORBA component through two forms of external interfaces - a **home** interface and one or more **application** interfaces. Home interfaces support operations which allow the client to obtain references to an application interface which the component implements.

From the client's perspective, the home supports two design patterns - factories for creating new objects and finders for existing objects. These are distinguished by the presence of a **primaryKey** parameter in the home IDL.

- if a **primaryKey** is defined, the home supports both factories and finders and the client may use both.
- if a **primaryKey** is not defined, the home supports only the factory design pattern and the client must create new instances.

Two forms of clients are supported by the CORBA component model:

- Component-aware clients - These clients know they are making requests against a component (as opposed to an ordinary CORBA object) and can therefore avail themselves of unique component function, e.g. navigation among multiple interfaces and component type factories.
- Component-unaware clients - These clients do not know that the interface they are making requests against is implemented by a CORBA component so they can only invoke functions supported by an ordinary CORBA object, e.g. looking up a name in a Naming or Trader service, searching for a particular type of factory using a factory finder, etc.

62.5.1 Component-aware Clients

Clients that are defined using the IDL extensions in Chapter 61, "Component Model" are referred to as **component-aware** clients. Such clients can avail themselves of the unique features of CORBA components which are not supported by ordinary CORBA objects. The interaction between these clients and a CORBA component are outlined in the following sections. A **component-aware** client interacts with a component through one or more CORBA interfaces:

- the equivalent interface implied by the **component** IDL declaration,
- zero or more supported interface declared on the **component** specification.
- zero or more interfaces defined by the **provides** clauses in the **component** definition,
- the home interface which supports factory and finder operations

Furthermore a component-aware client locates those interfaces using the **Components::HomeFinder** or a naming service. The starting point for client interactions with the component is the **resolve_initial_references** operation on **CORBA::ORB** which provides the initial set of object references.

62.5.1.1 *Initial References*

Initial references for all services used by a component client are obtained using the **CORBA::ORB::resolve_initial_references** operation. This operation currently supports the following references required by a component client:

- Name Service (“**NameService**”)
- Transaction Current (“**TransactionCurrent**”)
- Security Current (“**SecurityCurrent**”)
- Notification Service (“**NotificationService**”)
- Interface Repository (“**InterfaceRepository**”) for DII clients.
- Home Finder (“**ComponentHomeFinder**”)

The client uses **ComponentHomeFinder** (defined in Section 61.8, “Home Finders,” on page 61-62) to obtain a reference to the **HomeFinder** interface.

62.5.1.2 *Factory Design Pattern*

For factory operations, the client invokes a **create** operation on the home. Default create operations are defined for each category of CORBA components for which code can be automatically generated. These operations return an object of type **CORBA::Component** which must be narrowed to the specific type. Alternatively, the component designer may specify custom factories as part of the **component** definition to define a type-specific signature for the create operation. Because these operations are defined in IDL, operation names can be chosen by the component designer. All that is required is that the operations return an object of the appropriate type.

A client using the factory design pattern uses the **HomeFinder** to locate the component factory (**CCMHome**) by interface type. The **HomeFinder** returns a type-specific factory reference which can then be used to create new instances of the component interface. Once created, the client makes operation requests on the reference representing the interface. This is illustrated by the following code fragment below:

```

// Resolve HomeFinder
org.omg.CORBA.Object objref =
orb.resolve_initial_references("ComponentHomeFinder");

ComponentHomeFinder ff =
ComponentHomeFinderHelper.narrow(objref);

org.omg.CORBA.Object of =
ff.find_home_by_type(AHomeHelper.id());

AHome F = AHomeHelper.narrow (of);
org.omg.Components.ComponentBase AInst = F.create();
A Areal = AHelper.narrow (AInst);

// Invoke Application Operation
answer = A.foo(input);

```

62.5.1.3 Finder Design Pattern

A component-aware client wishing to use an existing component instance (rather than create a new instance) uses a **finder** operation. Finders are supported for entity components only. Client's may use the **HomeFinder** as described in Section 61.8, "Home Finders to locate the component's home or they may use CORBA naming to look up a specific instance of the home by symbolic name.

A client using the finder design pattern uses the **CosNaming::NamingContext** interface to lookup a symbolic name. The naming service returns an object reference of the type previously bound. The client then makes operation requests on the reference representing the interface. This is illustrated by the following code fragment below:

```

org.omg.CORBA.Object objref =
orb.resolve_initial_references("NamingService");

NamingContext ncRef = NamingContextHelper.narrow(objref);

// Resolve the Object Reference in Naming
NameComponent nc = new NameComponent("A","");
NameComponent path[] = {nc};
A aRef = AHelper.narrow(ncRef.resolve(path));

// Invoke Application Operation
answer = A.foo(input);

```

62.5.1.4 Transactions

A component-aware client may optionally define the boundaries of the transaction to be used with CORBA components. If so, it uses the CORBA transaction service to ensure that the active transaction is associated with subsequent operations on the CORBA component.

The client obtains a reference to **CosTransactions::Current** by using the **CORBA::ORB::resolve_initial_references** operation specifying an **ObjectID** of **"TransactionCurrent"**. This permits the client to define the boundaries of the transaction, i.e. how many operations will be invoked within the scope of the client's transaction. All operations defined for **Current** may be used as defined by the CORBA transaction service with the following exceptions:

- The **Control** object returned by **get_control** and **suspend** may only be used with **resume**.
- Operations on **Control** may raise the **NO_IMPLEMENT** exception with CORBA components.

*The **Control** interface in the CORBA transaction service supports accessors to the **Coordinator** and **Terminator** interfaces. The **Coordinator** is used to build object versions of XA resource managers. The **Terminator** is used to allow a transaction to be ended by someone other than the originator. Since neither function is within the scope of the demarcation subset of CORBA transactions used with CORBA components, we allow CORBA transaction services implementations used with CORBA components to raise the **NO_IMPLEMENT** exception.*

The following code fragment shows a typical usage:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references("TransactionCurrent");

Current txRef = CurrentHelper.narrow(objRef);
txRef.begin();
// Invoke Application Operation
answer = A.foo(input);
txRef.commit();
```

62.5.1.5 Security

A component-aware client uses the existing CORBA security mechanism to manage security for a CORBA component. There are two scenarios possible:

- Use of SSL for establishing client credentials
CORBA security today does not define a standard API for clients to use with SSL to set the credentials which will be used to authorize subsequent requests. The credentials must be set in a way which is proprietary to the client ORB.
- Use of SECIOP by the client ORB.

In this case, CORBA security does define an API and it must be used by the client to establish the credentials to be used to authorize subsequent requests.

Security processing for CORBA components uses a subset of CORBA security. For SECIOP, the client sets the credentials to be used with subsequent operations on the component by using operations on the **SecurityLevel2::PrincipalAuthenticator**. The client obtains a reference to **SecurityLevel2::Current** by using the **CORBA::ORB::resolve_initial_references** operation specifying an **ObjectID** of

“SecurityCurrent”. This permits the client to access the **PrincipalAuthenticator** interface to associate security credentials with subsequent operations. The following code fragment shows a typical usage:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references("SecurityCurrent");

org.omg.SecurityLevel2.PrincipalAuthenticator secRef =
org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow
(objRef);

secRef.authenticate(...);

// Invoke Application Operation
answer = A.foo(input);
```

62.5.1.6 Events

Component-aware clients wishing to **emit** or **consume** events use the component APIs defined in Chapter 61, “Component Model”. Alternatively, they may use CORBA notification directly and conform to the subset supported by CORBA components (see Section 62.5.2.6 for details).

62.5.2 Component-unaware Clients

CORBA components can also be used by clients who are unaware that they are making requests against a component. Such clients can see only a single interface (the supported interface of a component) and do not support navigation.

62.5.2.1 Initial References

Component-unaware clients obtain initial references using existing CORBA mechanisms, viz. **CORBA::ORB::resolve_initial_references**. It is unlikely, however, that this mechanism would be used to obtain a reference to the **HomeFinder**.

62.5.2.2 Factory Design Pattern

The factory design pattern can be used by component-unaware clients only if the supported interface has application operations defined. This permits existing CORBA objects to be easily converted to CORBA components, transparently to their existing clients. The following techniques can be used:

- The reference to a factory finder (typically the **CosLifeCycle::FactoryFinder**) can be stored in the Naming or Trader service and looked up by the client before creating the instance.
- A reference to the home interface can be obtained from the Naming service.
- The reference to the home interface can be obtained from a Trader service.

- After locating a factory finder, the factory can be located using the existing **find_factories** operation or by using the new **find_factory** operation on the **CosLifeCycle::FactoryFinder** interface. The **find_factory** is defined in Section 11.3.1 on page 397.

*The current **CosLifeCycle** **find_factories** operation returns a sequence of factories to the client requiring the client to choose the one which will create the instance. To allow the server (i.e. the **FactoryFinder**) to make the selection, we also add a new **find_factory** operation to **CosLifeCycle** which allows the server to choose the “best” factory for the client request based on its knowledge of workload, etc.*

A **FactoryFinder** will return an **Object**. A component-unaware client may expect to narrow this to **CosLifeCycle::GenericFactory** and use the generic create operation. For this reason, we allow the default creation operation on home to return a **GenericFactory** interface. This is fully described in Section 61.7, “Homes.

- A stringified object reference can be retrieved from a file known by the component-unaware client.

Once a reference to the home has been obtained, the client can create component instances and make operation requests on the component. Each component exports at least one IDL interface. A supported interface must be used by the client to invoke the component’s application operations. Provided interfaces cannot be located using the factory design pattern.

62.5.2.3 Finder Design Pattern

A component-unaware client can use CORBA naming to locate an existing **entity** component. Unlike the factory design pattern, the name to be looked up by the client can be either a supported interface or any of the provided interfaces. The following techniques can be used:

- A symbolic name associated with the component’s home can be looked up in a Naming service to make an invocation of the finder operations.
- Alternatively, the reference to the home interface can be obtained from a Trader service.
- the finder operation can be invoked on the **entity** component to return a reference to the client.

62.5.2.4 Transactions

This is the same as component-aware clients (See Section 62.5.1.4). However, the possibility of the **NO_IMPLEMENT** exception being raised for operations on **Control** may have a more serious impact, since the component-unaware client may not be expecting that to happen.

62.5.2.5 Security

This is the same as component-aware clients (See Section 62.5.1.5).

62.5.2.6 Events

Component-unaware clients wishing to **emit** or **consume** events must use the equivalent CORBA notification interfaces and stay within the subset supported by CORBA components (see Section 62.2.8 for details). This is illustrated by the following code fragment:

```
org.omg.CORBA.Object objref =
orb.resolve_initial_references("NotificationService");

org.omg.CosNotifyChannelAdmin.EventChannelFactory evfRef =
org.omg.EventChannelFactoryHelper.narrow(objRef);

// Create an Event Channel
org.omg.CosNotifyChannelAdmin.EventChannel evcRef =
evfRef.create_channel(...);

// Obtain a SupplierAdmin
org.omg.CosNotifyChannelAdmin.SupplierAdmin publisher =
evcRef.new_for_suppliers (...);

// And a ConsumerProxy
org.omg.CosNotifyComm.ProxyConsumer proxy =
publisher.obtain_notification_push_comsumer (...);

// Publish a structured event
proxy.push_structured_event(...);
```

This chapter describes the integration of CORBA components with Enterprise JavaBeans.

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

64.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Introduction”	64-172
“Enterprise JavaBeans Compatibility Objectives and Requirements”	64-174
“CORBA Component views for EJBs”	64-175
“EJB views for CORBA Components”	64-183
“Comparing CCM and EJB”	64-190

64.1 Introduction

This chapter describes how an Enterprise JavaBeans (EJB) component can be used by CORBA clients, including CORBA components. The EJB will have a CORBA component style remote interface that is described by CORBA IDL (including the component extensions).

This chapter also describes how a CORBA component can be used by a Java client, including an Enterprise JavaBeans component. The CORBA component will have an EJB style remote interface that is defined following the Enterprise JavaBeans specification.

The concepts in this chapter follow in the same prescription for interworking as laid out in Chapter 17 of the CORBA CORE specification where it is discussed as follows:

How interworking can be practically achieved is illustrated in an Interworking Model, shown in Figure 64-1 on page 173. It shows how an object in Object System B can be mapped and represented to a client in Object System A. From now on, this will be called a B/A mapping. For example, mapping a CORBA Component Model object to be visible to an EJB client is a CCM/EJB mapping.

On the left is a client in object system A, that wants to send a request to a target object in system B, on the right. We refer to the entire conceptual entity that provides the mapping as a bridge. The goal is to map and deliver any request from the client transparently to the target.

To do so, we first provide an object in system A called a View. The View is an object in system A that presents the identity and interface of the target in system B mapped to the vernacular of system A, and is described as an A View of a B target. The View exposes an interface, called the View Interface, which is isomorphic to the target's

interface in system B. The methods of the View Interface convert requests from system A clients into requests on the target's interface in system B. The View is a component of the bridge. A bridge may be composed of many Views.

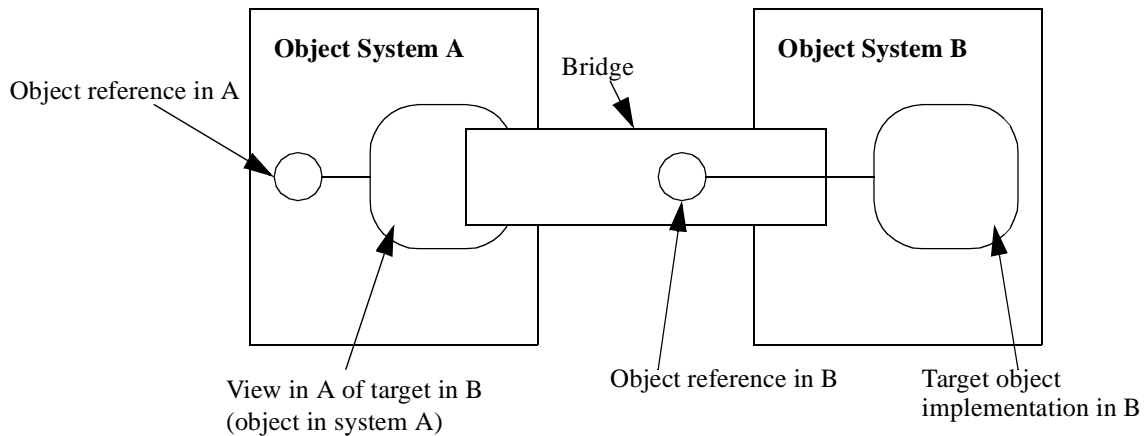


Figure 64-1 B/A Interworking Model

The bridge maps interface and identify forms between different object systems. Conceptually, the bridge holds a reference in B for the target (although this is not physically required). The bridge must provide a point of rendezvous between A and B, and may be implemented using any mechanism that permits communication between the two systems (IPC, RPC, network, shared memory, and so forth) sufficient to preserve all relevant object semantics.

The client treats the View as though it is the real object in system A, and makes the request in the vernacular request form of system A. The request is translated into the vernacular of object system B, and delivered to the target object. The net effect is that a request made on an interface in A is transparently delivered to the intended instance in B.

The Interworking Model works in either direction. For example, if system A is EJB, and system B is CCM, then the View is called the EJB View of the CCM target. The EJB View presents the target's interface to the EJB client. Similarly if system A is CCM and system B is EJB, then the View is called the CCM View of the EJB target. The CCM View presents the target's interface to the CCM client.

64.2 *Enterprise JavaBeans Compatibility Objectives and Requirements*

The objective is to allow the creation of distributed applications which mix CORBA components running in CORBA component servers with EJB components running in an EJB technology-based server. This objective allows a developer to create an application by reusing existing components of either kind.

This requires development time and runtime translations between the CORBA component and EJB domains provided by mediated bridges. It also requires that:

- A CORBA component view for an EJB comply with the EJB to CORBA mapping specification. In particular, this requires that:
 - An EJB definition be mapped to a CORBA component definition following the Java Language to IDL mapping plus the extensions to that mapping that are specified in this chapter.
 - Value objects of one kind (e.g. Keys for EJB) have counterpart value objects of the other kind.
 - CORBA components accessible via **CosNaming** have their EJB views accessible via **JNDI**, and vice versa.
- An EJB view for a CORBA component comply with the EJB specification.

An application is to be built using both EJB and CORBA components deployed in their respective containers. At component development time, EJB components are originally defined in Java and CORBA components are originally defined in IDL. When applications are assembled using both, the application assembly environment will most commonly dictate which model these components must present to developers. During application assembly, developers construct clients (which themselves may be components) that make use of components in the way most natural to the particular environment. Thus in a CORBA environment clients will expect to make use of both

the CCM model and the EJB model as CORBA components, and in an EJB environment, clients will expect to make use of both kinds as enterprise beans. All four combinations of clients and components are illustrated in Figure 64-2 on page 175.

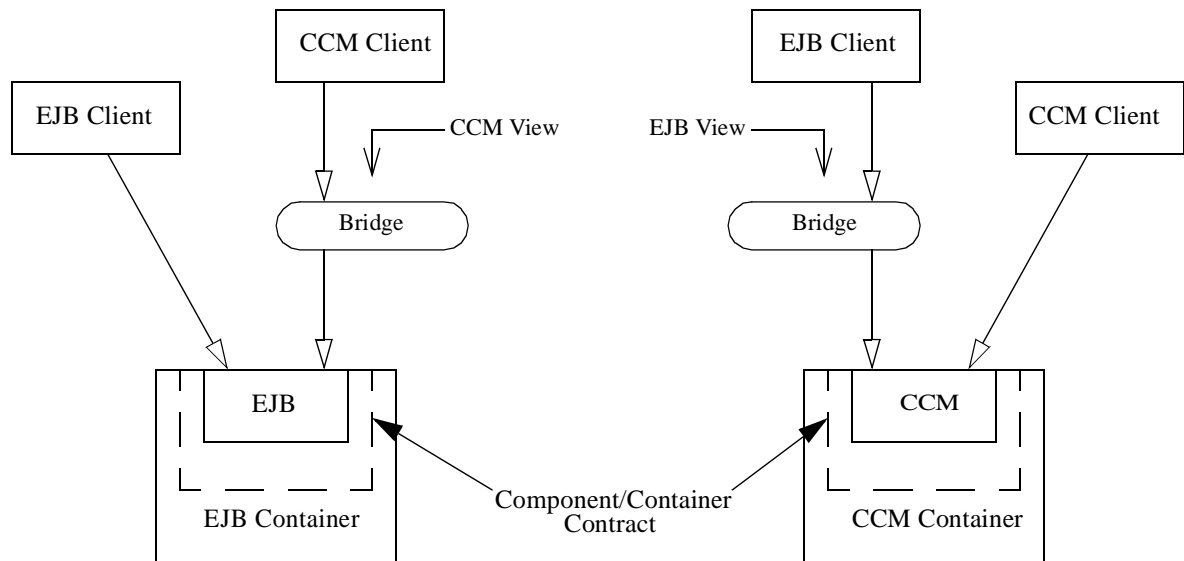


Figure 64-2 Interoperation in a mixed environment

In this scenario, components of one kind are made accessible to clients of another by way of two mechanisms: generation of bindings at development time and method translation at runtime. Thus, the containers provide an EJB view of a CORBA component and a CCM view of an EJB.

For application developers in a CORBA environment, EJBs specified in Java are mapped to CORBA IDL for use by CCM clients, and at runtime client calls on CCM methods are translated by a bridge into EJB methods. In effect, the EJBs *are* CORBA components.

For application developers in an EJB environment, CORBA components specified in IDL are mapped to Java interfaces for use by EJB clients, and at runtime client calls on EJB methods are translated by a bridge into CCM methods. In effect, the CORBA components *are* EJBs.

64.3 CORBA Component views for EJBs

This kind of view allows a CORBA client -- either a CORBA component or any piece of code that uses CORBA, and either component-aware or not -- to access an EJB as a CORBA component. To do this, two things are needed:

- A mapping of the definition of the existing EJB into the definition of a CORBA component. This mapping takes an EJB's RMI remote interface and home interface and produces an equivalent CORBA component definition.

- A translation, at run-time, of CORBA component requests performed by a CORBA client into EJB requests. This translation can be performed in terms of either straight delegation, or as an interpretation of a CORBA client request in terms of EJB requests.

64.3.1 Mapping of EJB to Component IDL definitions

An EJB definition includes the following EJB interfaces:

- An EJB home interface. This interface extends the pre-defined **EJBHome** interface.
- An EJB remote interface. This interface extends the pre-defined **EJBObject** interface.

Thus, for the purposes of this chapter, at least these EJB interfaces must be mapped into IDL in order to obtain a CORBA component definition of a view that a CORBA client can use to make requests on an existing EJB. An EJB home interface definition maps into a CORBA component's home definition, whose implied IDL inherits from **CCMHome**. This means that **EJBHome** is mapped into **CCMHome**. Likewise, an EJB remote interface definition maps into a basic CORBA component definition, whose implied IDL inherits from **CCMObject**. This means that **EJBObject** is mapped into **CCMObject**.

In addition, **EJBHome** and **EJBObject** make use of the following pre-defined EJB interfaces:

- The **HomeHandle** interface.
- The **Handle** interface.
- The **EJBMetaData** interface.

Handles are an EJB concept that has no direct counterpart in CORBA components. Thus, **HomeHandle** and **Handle** are not directly mapped into equivalent IDL.

*Notice that although Interoperable Object References (IORs) and the ORB provided operations that manipulate them (**string_to_object** and **object_to_string**) are conceptually similar to Handles, there are enough differences between IORs and Handles to preclude a mapping from Handles to IORs.*

Meta data is available to a CORBA client but not in the same form as that provided by **EJBMetaData**. Given that an EJB maps into a CORBA component, whose definition produces the meta data that a CORBA client expects, mapping **EJBMetaData** into equivalent IDL is not required.

64.3.1.1 Java Language to IDL Mapping

The reader is assumed to be familiar with the specification for the Java to IDL mapping, whose major aspects are repeated here for convenience.

- A Java interface is an RMI/IDL remote interface if it at least extends `java.rmi.Remote` and all of its methods throw `java.rmi.RemoteException`.
- get- and set- name pattern names are translated to IDL attributes.
- IDL generated methods have only **in** parameters (but these can include object references to remote objects, allowing reference semantics normally obtained by using parameters of type `java.rmi.Remote`).
- Java objects that inherit from `java.io.Serializable` or `java.io.Externalizable` are mapped to a CORBA valuetype. All object types appearing in RMI remotable interfaces must inherit from these interfaces or from `java.rmi.Remote`. EJB **Key** and **Handle** types must inherit from `java.io.Serializable`.
 - However, the mapping does NOT require that methods on such objects or constructors be mapped to corresponding IDL operations on **valuetypes** and **init** specifications. The developer is expected to select those methods which should be mapped to IDL operations, and the method signatures must meet the requirements of the mapping.
 - Objects which inherit from `java.io.Externalizable` or which implement `writeObject` are understood to perform custom marshalling and the corresponding custom marshallers must be created for the CORBA valuetype.
- Arrays are mapped to “boxed” CORBA **valuetypes** containing sequences because Java arrays are dynamic.
- Java exceptions are subclassable; IDL exceptions are not. Consequently a name pattern is used to map to IDL exceptions. The Java exception object is mapped to a CORBA valuetype. The CORBA valuetype has an inheritance hierarchy like that of the corresponding Java exception object.
- Some additional programming is required to define Java classes (including EJB implementations) that are accessible via RMI/IIOP. This is to account for the fact that IIOP does not support distributed garbage collection.

64.3.1.2 EJB to IDL mapping

In general, the CORBA component that results from mapping an EJB will support an interface that is the Java to IDL map of the Remote interface of the EJB. The mapping rules are as follows.

Mapping the Remote Interface

- An EJB’s remote interface maps to a definition of a basic CORBA component that supports the default interface. The form of the CORBA component definition is **component XXX supports XXXDefault**.
- An EJB’s remote interface declaration is used to create a **supports** declaration and the corresponding IDL for the primary interface of the CORBA component that the EJB maps to. The identifier of this supported interface on the component is

XXXDefault, where **XXX** is the name of the EJB remote interface. This generated interface is referred to as the *Default* interface of the component that the given EJB maps to.

- Each operation on the Remote interface is mapped under Java to IDL to an equivalent operation on the **XXXDefault** interface. Note that pairs of **getXXX** and **setXXX** methods in the EJB remote interface will be mapped to IDL attributes. Any exceptions thrown by a **getXXX** method is mapped to an exception in the **getRaises** clause of the mapped IDL attribute. Likewise, any exception thrown by a **setXXX** method is mapped to an exception in the **setRaises** clause of the mapped IDL attribute. The actual definitions of the exceptions thrown are mapped following the Java to IDL rules.

Mapping the Home Interface

- An EJB's home interface maps to a definition of a CORBA component home. The form of the CORBA component home definition is **home YYY manages XXX**, where **YYY** is the name of the EJB home interface. Mapping an EJB home into a CORBA component home requires the existence of meta data that links the EJB home to the EJB that it hosts. These meta data are obtained from the EJB's deployment descriptor. Thus **XXX** is the name of the EJB that the EJB home hosts, as it is given in the EJB deployment descriptor.
- The EJB home methods called **create** are mapped into home **factory** declarations in IDL. The actual names of each of the **factory** operations are produced following the rules for mapping Java names to IDL names in the Java to IDL specification. The Java parameters of the operation are mapped to their corresponding IDL types and names as defined by Java to IDL.
- An EJB Primary Key class is mapped to a CORBA **valuetype** using the mapping rules in Java to IDL. This **valuetype** will be declared in the IDL for the CORBA component home as the primary key **valuetype** for the component. The key **valuetype** will inherit from **Components::PrimaryKeyBase**. If an EJB home uses a primary key, then the form of the CORBA component home definition is **home YYY manages XXX primaryKey KKK**, where **KKK** is the name of the valuetype that the EJB primary key class maps to.
- The EJB home operation named **findByPrimaryKey** is mapped into the **find_by_primary_key(in <key-type> primaryKey)** operation on the component's implicit home interface.
- Finder and Creator EJB operations that return an RMI style object reference are mapped into Component IDL operations which return a CORBA Component Object Reference (**Components::CCMObject**).
- EJB home operations prefixed **find** whose return type is the type of the EJB hosted by the EJB home are mapped into component home **finder** operations in IDL. The actual names of each of the **finder** operations are produced following the rules for mapping Java names to IDL names in the Java to IDL specification. The Java parameters of the operation are mapped to their corresponding IDL types and names as defined by Java to IDL.

- Finder EJB operations that return a Java Enumeration are mapped into CORBA component operations which return an IDL Object Reference to an interface of type **Enumeration**. This interface is declared as:

```
module Components {

    interface Enumeration {
        boolean has_more_elements();
        CCMObject next_element();
    };
};
```

The Enumeration interface is just the RMI/IIOP image of the Java Enumeration class as defined in the JDK 1.1.6+. Sun has said that they intend to replace this with the JDK 1.2 (Java 2.0) Collections in a future version of the EJB specification. Subsequent to such a specification being issued, the CORBA components specification will be updated to correspond.

- In order for an EJB home definition that defines **findByPrimaryKey** to be successfully mapped onto a CORBA component home definition, it must define a **create** method that takes the primary key of the hosted EJB as its sole argument and returns an instance of the hosted EJB. This create method is mapped to **create(in <key-type> key)** on the CORBA component implicit home interface.

64.3.2 Translation of CORBA Component requests into EJB requests

A CORBA client that uses a CORBA component view on an EJB expects to be able to perform CORBA component requests on such a view. These requests need to be translated into EJB requests at run-time. This translation can be performed at the client-side, server-side, or a combination of the two. Table 64-1 lists the CORBA component operations that a CORBA client can perform requests on by interface, and it lists the corresponding EJB methods that these requests translate into, also by interface.

Notice that a CORBA client may use operations on object references such as **string_to_object** and **object_to_string** that may be considered as analogous to EJB **Handle** methods. However, these operations are not seen by the bridge since they are

Table 64-1 Translation of CCM operation requests into EJB method requests

CCM Interface	Operation called by client	EJB interface	Method invoked by bridge
CCMHome	ComponentDef get_component_def (); void remove_component (in CCMObject comp) raises (CCMException);	EJBHome	EJBMetaData getEJBMetaData () throws RemoteException; void remove (Handle handle) throws RemoveException, RemoteException;
<home-name>Explicit <home-name>Implicit	<name> createXXX (<arg-list>) raises (DuplicateKeyValue, InvalidKey); <name> findXXX (<arg-list>) raises (<exceptions>); <name> create (in <key-type> key) raises (DuplicateKeyValue, InvalidKey); <name> find_by_primary_key (in <key-type> key) raises (UnknownKeyValue, InvalidKey); void remove (in <key-type> key) raises (UnknownKeyValue, InvalidKey); <key_type> get_primary_key (in <name> comp);	<home-name> EJBHome EJBObject	<name> create (<arg-list>) throws CreateException, DuplicateKeyException; <name> findXXX (<arg-list>) throws <exceptions>; <name> create (Object primaryKey) throws CreateException, DuplicateKeyException; <name> findByPrimaryKey (<key-type> key) throws FinderException, ObjectNotFoundException; void remove (Object primaryKey) throws RemoveException, RemoteException; Object getPrimaryKey () throws RemoteException;

Table 64-1 Translation of CCM operation requests into EJB method requests

CCM Interface	Operation called by client	EJB interface	Method invoked by bridge
CCMObject	ComponentDef get_component_def (); CCMHome get_home (); PrimaryKeyBase get_primary_key (); void remove(); void configuration_complete () raises (InvalidConfiguration);	EJBHome EJBObject	EJBMetaData getEJBMetaData () throws RemoteException; EJBHome getEJBHome () throws RemoteException; Object getPrimaryKey () throws RemoteException; void remove () throws RemoveException, RemoteException; Translation performed by bridge is to raise the NO_IMPLEMENT exception
<name>	<res-type> <operation> (<arg-list>) raises (<exceptions>); <res-type> getXXX () throws <exceptions>; void setXXX (<arg-list>) throws <exceptions>;	<name>	<res-type> <operation> (<arg-list>) throws <exceptions>; <res-type> getXXX () throws <exceptions>; void setXXX (<arg-list>) throws <exceptions>;

performed on the ORB and thus no translation for these operations on the part of the bridge is required.

The following restrictions apply:

- **create (in <key_type> key)** on the component implicit home interface can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **remove (in <key_type> key)** on the component implicit home interface can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **get_primary_key** on the component implicit home and on **CCMObject** can only be validly invoked by a CORBA client if the underlying EJB home declares the **findByPrimaryKey** operation.
- **configuration_complete** on **CCMObject** is not translated by the bridge, a request on this operation by a CORBA client raises the **NO_IMPLEMENT** exception.

64.3.3 CORBA Component view Example

In this section we show a simple EJB together with the corresponding Component IDL. Note that the EJB deployment metadata is needed to generate the IDL; this is because the metadata binds together the Remote interface and the Home interface.

Below are the remote interfaces of the EJB.

```
package example;

class CustInfo implements java.io.Serializable
{
    public int custNo;
    public String custName;
    public String custAddr;
};

class CustBal implements java.io.Serializable
{
    public int custNo;
    public float acctBal;
};

interface CustomerInquiry extends javax.ejb.EJBObject
{
    CustInfo getCustInfo(int iCustNo)
        throws java.rmi.RemoteException;
    CustBal getCustBal(int iCustNo)
        throws java.rmi.RemoteException;
};

interface CustomerInquiryHome extends javax.ejb.EJBHome
{
    CustomerInquiry create()
        throws java.rmi.RemoteException;
};
```

Below are the contents of the descriptor classes as they might be expressed in an equivalent XML document.

```
<ejb-jar>
  <session>
    <description>
    </description>
    <ejb-name> CustomerInquiry </ejb-name>
    <home> example.CustomerInquiryHome </home>
    <remote> example.CustomerInquiry </remote>
    <ejb-class> example.CustomerInquiryBean </ejb-class>
    <session-type> Stateful </session-type>
  </session>
</ejb-jar>
```

The EJB is a session bean, and in this case, its **create** operation requires no parameters. The two operations take a key value and return values to the caller. The EJB implementation will use **JDBC** to retrieve the information to be returned by the operations on the **CustomerInquiry** EJB.

The serializable value classes are translated by RMI/IIOP into CORBA concrete **valuetypes** as follows:

```
valuetype CustInfo {
    public long custNo;
    public ::CORBA::WStringValue custName;
    public ::CORBA::WStringValue custAddr;
};
```

```
valuetype CustBal {
    public long custNo;
    public float custBal;
};
```

The information in the deployment descriptor and the home and remote interface declarations is introspected and used to generate the following IDL.

```
interface CustomerInquiryDefault {
    CustInfo getCustInfo(in long iCustNo);
    CustBal getCustBal(in long iCustNo);
};

component CustomerInquiry supports CustomerInquiryDefault {};

home CustomerInquiryHome manages CustomerInquiry {
    factory create();
};
```

64.4 EJB views for CORBA Components

This kind of view allows a Java client -- either an EJB or any other piece of Java code -- to access a CORBA component as an EJB. To do this, two things are needed:

- A mapping of the Component IDL definition of a CORBA component into an EJB definition. This mapping only considers that portion of the Component IDL language that has a counterpart in the EJB specification language and it ignores the rest.
- A translation, at run-time, of EJB requests performed by a Java client into CORBA component requests.

64.4.1 Mapping of Component IDL to Enterprise JavaBeans specifications

The portion of the Component extensions to the IDL language that can be mapped to the EJB specification language is denoted by the following subset of the Component extensions to IDL grammar.

```

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

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

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

<component_body> ::= <component_export>*

<component_export> ::= <attr_dcl> ";;"

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

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

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

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

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

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

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

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

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

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

<home_header> ::= "home" <identifier> "manages" <scoped_name> [
    <primary_key_spec> ]

<primary_key_spec> ::= "primaryKey" <scoped_name>

<home_body> ::= "{" <home_export>* "}"

<home_export> ::= <factory_dcl> ";;" | <finder_dcl> ";;"

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

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

```


The rules for mapping a CORBA component definition into an EJB definition are defined in the following sections. Where appropriate, these rules rely on the standard IDL to Java mapping.

Mapping the component definition

- A basic CORBA component definition is mapped to an EJB remote interface definition.
- The name of the EJB remote interface is the name of the basic CORBA component in the Component IDL definition.
- For each operation defined in each interface that the CORBA component **supports**, a method definition will be included in the EJB remote interface that the CORBA component maps to. That is, the EJB to which the basic CORBA component maps defines all the supported operations defined by the basic CORBA component.
- The signatures of the CORBA component operations are mapped to signatures of EJB remote interface methods following the IDL to Java mapping rules.
- For each attribute **XXX** that the CORBA component defines, the corresponding EJB remote interface defines a pair of **getXXX** and **setXXX** methods, where **XXX** is the name of the given attribute. If the attribute definition includes a **getRaises** exception clause, then the corresponding **getXXX** method definition in the EJB remote interface will include a throws exception clause. Likewise, if the attribute definition includes a **setRaises** exception clause, then the corresponding **setXXX** method definition in the EJB remote interface will include a throws exception clause.
- Exceptions raised by CORBA component definition operations and attributes are mapped to exceptions thrown by EJB method definitions using the standard IDL to Java mapping rules.

Mapping the Component Home definition

- A CORBA component's home definition is mapped to an EJB home's remote interface definition. That is a definition of the form **home XXX manages YYY [primaryKey KKK]** is mapped to an EJB home interface with name **xxx**.
- The methods defined by the EJB home remote interface include the implicit as well as the explicit methods of the CORBA component's home definition.
- Implicit CORBA component home operations are mapped to EJB home remote interface methods as follows:
 - `<component_type> create (in <key_type> key) raises (Components::DuplicateKeyValue, Components::InvalidKey);` maps to `<component_type> create (<key_type> key) throws DuplicateKeyException, CreateException.`

- `<component_type> find_by_primary_key (in <key_type> key)` raises `(Components::UnknownKeyValue, Components::InvalidKey)`; maps to `<component_type> findByPrimaryKey(<key_type> key)` throws `ObjectNotFoundException, FinderException`.
- `void remove (in <key_type> key)` raises `(Components::UnknownKeyValue, Components::InvalidKey)`; maps to the `remove` by key method defined in **EJBHome**.
- `<key_type> get_primary_key (in <component_type> comp)`; has no counterpart in an EJB home definition. Given that **EJBObject** already defines `getPrimaryKey`, it is not necessary to map `get_primary_key` on the implicit home to an EJB home operation.
- Explicit CORBA component basic home operations are mapped to EJB home remote interface methods as follows:
 - A **factory** operation maps to an overloaded **create** method with the corresponding arguments and exceptions.
 - A **finder** operations maps to a **find<identifier>** method with the corresponding arguments and exceptions, where **<identifier>** is the name of the **finder** operation.
 - The signatures of **factory** and **finder** operations are mapped to signatures of EJB home interface methods following the IDL to Java mapping rules.
- A **valuetype** that is used to define the primary key of a CORBA component home is mapped to a Java class under the rules of the standard IDL to Java mapping. In addition, such a Java class is defined to extend **java.io.Serializable**.

64.4.2 Translation of EJB requests into CORBA Component requests

A Java client that uses an EJB view on a CORBA component expects to be able to perform EJB requests on such a view. These requests need to be translated into CORBA component requests at run-time. This translation can be performed at the client-side, the server-side, or a combination of the two. Table 64-2 lists the EJB

methods that a Java client can perform requests on by interface, and it lists the corresponding CORBA component operations that these requests translate into, also by interface.

Table 64-2 Translation of EJB method requests into CCM operation requests

EJB Interface	Method called by client	CCM interface	Operation called by bridge
EJBHome	EJBMetaData getEJBMetadata () throws RemoteException; void remove (Handle handle) throws RemoveException, RemoteException; void remove (Object primaryKey) throws RemoveException, RemoteException; HomeHandle getHomeHandle () throws RemoteException;	CCMHome <home-name>Implicit	Translation performed by bridge does not call a CCM standard operation void remove_component (in CCMObject comp) raises (CCMException); void remove (in <key-type> key) raises (UnknownKeyValue, InvalidKey); Translation performed by bridge does not call a CCM standard operation
<home-name>	<name> create (<arg-list>) throws CreateException, DuplicateKeyException; <name> findByXXX (<arg-list>) throws <exceptions>; <name> findByPrimaryKey (<key-type> key) throws FinderException, ObjectNotFoundException;	<home-name>Explicit <home-name>Implicit	<name> createXXX (<arg-list>) raises (DuplicateKeyValue, InvalidKey); <name> findXXX (<arg-list>) raises (<exceptions>); <name> find_by_primary_key (in <key-type> key) raises (UnknownKeyValue, InvalidKey);
EJBObject	EJBHome getEJBHome () throws RemoteException; Object getPrimaryKey () throws RemoteException; void remove () throws RemoveException, RemoteException; boolean isIdentical (EJBObject object) throws RemoteException; Handle getHandle () throws RemoteException;	CCMObject CORBA::Object	CCMHome get_CCM_home (); PrimaryKeyBase get_primary_key (); void remove (); boolean is_equivalent (); Translation performed by bridge does not call a CCM standard operation

Table 64-2 Translation of EJB method requests into CCM operation requests

EJB Interface	Method called by client	CCM interface	Operation called by bridge
<name>	<res-type> <operation> (<arg-list>) throws <exceptions>; <res-type> getXXX () throws <exceptions>; void setXXX (<arg-list>) throws <exceptions>;	<name>	<res-type> <operation> (<arg-list>) raises (<exceptions>); <res-type> get_XXX () raises (<exceptions>); <res-type> set_XXX () raises (<exceptions>);
EJBMetadata	EJBHome getEJBHome () throws RemoteException; Class getHomeInterfaceClass () throws RemoteException; Class getRemoteInterfaceClass () throws RemoteException; Class getPrimaryKeyClass () throws RemoteException; boolean isSession () throws RemoteException; boolean isStatelessSession() throws RemoteException		Translation performed by bridge on all these invocations does not call a CCM standard operation

In addition, the EJB programming model allows a Java client to:

- Locate EJB homes and distinguished EJB objects via **JNDI**
- Demarcate transactions via a **UserTransaction** object, after locating this object via **JNDI**

These requests are translated into similar requests provided by the CORBA component programming model, as follows:

- Location of home and EJB objects requires the definition of a mapping of JNDI to the COSNaming service. It also requires the mapping of a COSNaming name space into a JNDI name space.
- Transaction demarcation requires the definition of a mapping of JTA to the CORBA transaction service. It also requires that a JNDI name space location be populated with an object that implements **UserTransaction** and that maps to the corresponding CORBA transaction service object.

64.4.3 Example

We show a simple CORBA component definition and its corresponding EJB mapping. The basic CORBA component **Account** is defined in terms of a regular IDL interface **AccountOps**. The home **AccountHome** is defined to manage **Account** and to use a primary key.

```
interface AccountOps {
    void debit( in double amt ) raises (NotEnoughFunds);
    void credit( in double amt );
};

component Account supports AccountOps {
    readonly attribute double balance;
};

valuetype AccountKey {
    public long acctNo;
};

home AccountHome manages Account primaryKey AccountKey {
    finder largeAccount( double threshold );
};
```

The following EJB definition is derived from the definition of **Account** and its home.

```

public interface Account extends javax.ejb.EJBObject {
    public void debit( double amount )
        throws NotEnoughFunds, java.rmi.RemoteException;
    public void credit( double amount ) throws java.rmi.RemoteException;
    public double getBalance() throws java.rmi.RemoteException;
};

public class AccountKey implements java.io.Serializable {
    public long acctNo;
    public AccountKey( long k ) { acctNo = k; }
};

public interface AccountHome extends javax.ejb.EJBHome {
    public Account create( AccountKey key )
        throws DuplicateKeyException, CreateException,
        java.rmi.RemoteException;
    public Account findByPrimaryKey( Account key )
        throws ObjectNotFoundException, FinderException,
        java.rmi.RemoteException;
    public Account findByLargeAccount( double threshold )
        throws java.rmi.RemoteException;
};

```

64.5 Comparing CCM and EJB

The following series of tables summarized the component APIs for Enterprise Java Beans (EJB 1.1) and Basic CORBA Components. The tables are organized as follows:

1. The home interfaces that define the remote access protocols for creating or finding EJBs or CORBA components (Section 64.5.1).
2. The component interfaces that define the remote access protocols for invoking business operations on EJBs or CORBA components (Section 64.5.2).
3. The callback interfaces that the CORBA component or EJB programmer must implement (Section 64.5.3).
4. The Context interfaces that provide the component developer access to container-provided services (Section 64.5.4).
5. The Transaction interface that supports bean-managed or component-managed transactions (Section 64.5.5).
6. The metadata interfaces that support access to component metadata (Section 64.5.6).

64.5.1 The Home Interfaces

Table 64-3 compares the home interfaces and operations which make up the EJB and CORBA component models. In EJB, the **EJBHome** object is created by the EJB container provider's tools and provides implementations for methods of the base class and delegates factory or finder methods on a derived class (**<name>Home**) to similarly named methods on the bean itself (**<name>Bean**).

In the CORBA component model, homes are defined as righteous CORBA objects and the associated factory or finder methods are generated as operations on the home and the component developer implements these directly so the container need not provided delegation support. The component developer may not even need to provide implementations for the default factory and finder operations if sufficient information is provided with the component's definition.

For CORBA clients to use EJB implementations, the container provider must externalize **EJBHome** to the CORBA client as a CORBA component home. This is accomplished by extensions to the Java to IDL mapping defined in Chapter 8. For EJB clients to access CORBA component homes, the container provider must create an **EJBHome** object that serves as a bridge between equivalent operations on **EJBHome** and the CORBA component home. This bridge is also described in Chapter 8.

Table 64-3 Comparing the home interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components	
Interface	EJBHome extends java.rmi.Remote	CCMHome	
Operation	public EJBMetaData get EJBMetaData () throws java.rmi.RemoteException	ComponentDef get_component_def ();	CORBA IR supports more metadata
	public HomeHandle getHomeHandle() throws java.rmi.RemoteException		CORBA::object_to_string provides same function
	public void remove (HomeHandle handle) throws java.rmi.RemoteException, RemoveException	void remove_component (in CCMObject component) raises (CCMException);	CORBA references instead of handles REMOVE_ERROR is minor code
	public void remove (java.lang.Object primaryKey) throws java.rmi.RemoteException, RemoveException		similar operation is defined on <home>Implicit for Homes with primaryKey
Interface	HomeHandle extends java.io.Serializable		CORBA reference used for handle
	public EJBHome getEJBHome() throws java.rmi.RemoteException		CORBA::string_to_object
Module	<session-name>	<session-home>	
Interface	<session>home extends EJBHome	<session-home>::CCMHome, <session-home>Implicit, <session-home>Explicit	

Table 64-3 Comparing the home interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Operation	public <session-name>Remote create (<arg-type> <arg-list>) throws CreateException	<session-component> create ();	Generated operation Inherited from <home>Implicit
Module	<entity-name>	<entity-home>	
Interface	<entity>home extends EJBHome	<entity-home>::CCMHome, <entity-home>Implicit, <entity-home>Explicit	
Operation	public <entity-name>Remote create (<arg-type> <arg-list>) throws CreateException, DuplicateKeyException	<entity-component> create () raises (InvalidKey, DuplicateKey);	Generated operation Inherited from <home>Implicit
	public <entity-name>Remote findByPrimaryKey (<arg-type> <arg-list>) throws FinderException, ObjectNotFoundException	<entity-component> find (in <key-type> primaryKey) raises (InvalidKey, UnknownKeyType);	Generated operation Inherited from <home>Implicit
	public <entity-name>Remote find<method> (<arg-type> <arg-list>) throws FinderException, ObjectNotFoundException	<entity-component> <find-method> (in <arg-type> <arg-list>) raises (<exceptions>);	Specified operation Inherited from <home>Explicit

64.5.2 The Component Interfaces

Table 64-4 compares the component interfaces and operations which make up the EJB and CORBA component models. In EJB, the **EJBObject** object is created by the EJB container provider's tools and provides implementations for methods of the base class and delegates business methods to a derived class (**<name>Remote**).

In the basic CORBA component model, components are defined as righteous CORBA objects and the associated business methods are defined as operations on a supported interface and the component developer implements these directly so the container need not provided delegation support.

For CORBA clients to use EJB implementations, the container provider must externalize **EJBObject** to the CORBA client as a CORBA component. This is accomplished by extensions to the Java to IDL mapping defined in Chapter 8. For EJB clients to access CORBA components, the container provider must create an

EJBObject implementation that serves as a bridge between business methods on **EJBObject** and the basic CORBA component's supported interface. This bridge is also described in Chapter 8.

Table 64-4 Comparing the remote interfaces of EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components	
Interface	EJBObject extends java.rmi.Remote	CCMObject	
Operation	public EJBHome getEJBHome() throws java.rmi.RemoteException	CCMHome get_home();	
	public java.lang.Object primaryKey getPrimaryKey() throws java.rmi.RemoteException		operation defined on <entity>home
	public void remove (Handle handle) throws java.rmi.RemoteException, RemoveException	void remove() raises (CCMException);	CORBA references instead of handles; REMOVE_ERROR is minor code
	public Handle getHandle() throws java.rmi.RemoteException		CORBA::object_to_string
	public boolean isIdentical (EJBObject obj) throws java.rmi.RemoteException	boolean is_equivalent(in Object obj);	
Interface	Handle extends java.io.Serializable		CORBA reference used for handle
	public EJBObject getEJBObject() throws java.rmi.RemoteException		CORBA::string_to_object
Module	<session-bean>	<session-component>	
Interface	<session>Remote extends EJBObject	<session>::CCMObject	
	<res-type> <operation> (<arg-type> <arg-list>) throws <exceptions>	<res-type> <operation> (in <arg-type> <arg-list>) raises (<exceptions>);	business methods
Module	<entity-bean>	<entity-component>	
Interface	<entity>Remote extends EJBObject	<entity>::CCMObject	
	<res-type> <operation> (<arg-type> <arg-list>) throws <exceptions>	<res-type> <operation> (in <arg-type> <arg-list>) raises (<exceptions>);	business methods

64.5.3 The Callback Interfaces

Table 64-5 summarizes the callback interfaces the EJB programmer or basic CORBA component programmer must implement. The EJB interfaces are specified as Java interfaces in accordance with the EJB 1.1 specification dated June 28, 1999. The CCM interfaces are specified in IDL as defined in this specification.

Table 64-5 Comparing EJB and CCM Callback Interfaces

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components::Basic	
Interface	EnterpriseBean	EnterpriseComponent	
Interface	SessionBean extends EnterpriseBean	TransientComponent::EnterpriseComponent	
Operation	public void setSessionContext (SessionContext ctx) throws EJBException	void set_transient_context (in TransientContext ctx) raises (CCMException);	
	public void ejbActivate () throws EJBException	void ccm_activate () raises (CCMException);	
	public void ejbPassivate () throws EJBException	void ccm_passivate () raises (CCMException);	
	public void ejbRemove () throws EJBException	void ccm_remove () raises (CCMException);	
Interface	<name>Bean extends SessionBean		Home operations are not delegated in CCM.
Operation	public void ejbCreate (<Arg-type> <arg-list>) throws CreateException, EJBException		Implemented on home, CREATE_ERROR is minor code
Interface	SessionSynchronization	TransientSynchronization	
Operation	public void afterBegin () throws EJBException	void after_begin () raises (CCMException);	
	public void beforeCompletion() throws EJBException	void before_completion () raises (CCMException);	
	public void afterCompletion (boolean committed) throws EJBException	void after_completion (in boolean committed) raises (CCMException);	
Interface	EntityBean extends EnterpriseBean	PersistentComponent::EnterpriseComponent	
Operation	public void setEntityContext (EntityContext ctx) throws EJBException	void set_persistent_context (in PersistentContext ctx) raises CCMException;	
	public void unsetEntityContext () throws EJBException	void unset_persistent_context () raises (CCMException);	
	public void ejbActivate () throws EJBException	void ccm_activate () raises (CCMException);	

Table 64-5 Comparing EJB and CCM Callback Interfaces

Construct	EJB Form	CCM Form	Notes
	public void ejbLoad () throws EJBException	void ccm_load () raises (CCMException);	
	public void ejbStore () throws EJBException	void ccm_store() raises (CCMException);	
	public void ejbPassivate () throws EJBException	void ccm_passivate () raises (CCMException);	
	public void ejbRemove () throws RemoveException, EJBException	void ccm_remove () raises (CCMException);	REMOVE_ERROR is a minor code
Interface	<name>Bean extends EntityBean		Home operations are not delegated in CCM.
Operation	public <key-type> ejbcreate (<Arg-type> <arg-list>) throws CreateException, DuplicateKeyException, EJBException		Implemented on home, CREATE_ERROR and DUPLICATE_KEY are minor codes
	public void ejbPostCreate () throws CreateException, DuplicateKeyException, EJBException		post_create not required in CCM due to CORBA identity model
	public <key-type> findByPrimaryKey (<Arg-type> <arg-list>) throws FinderException, NoSuchEntityException, ObjectNotFoundException, EJBException		Implemented on home, FIND_ERROR, NO_SUCH_ENTITY and OBJECT_NOT_FOUND are minor codes
	public <key-type> find<method> (<Arg-type> <arg-list>) throws FinderException, NoSuchEntityException, ObjectNotFoundException, EJBException		Implemented on home, FIND_ERROR, NO_SUCH_ENTITY and OBJECT_NOT_FOUND are minor codes

64.5.4 The Context Interfaces

The context interfaces summarized in Table 64-6 provide accessors to services provided by the component container. The are used by the component developer when these service are required.

Table 64-6 Comparing the EJB and CCM Context Interfaces

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	Components::Basic	
Interface	EJBContext	CCMContext	
Operation	public java.security.Principal getCallerPrincipal()	Principal get_caller_principal();	

Table 64-6 Comparing the EJB and CCM Context Interfaces

Construct	EJB Form	CCM Form	Notes
	public EJBHome getEJBHome()	CCMHome get_CCM_home();	
	public boolean getRollbackOnly() throws java.lang.IllegalState	boolean get_rollback_only() raises (IllegalState);	
	public javax.transaction.UserTransaction getUserTransaction () throws java.lang.IllegalState	Transaction::UserTransaction get_user_transaction () raises (IllegalState);	
	public boolean isCallerInRole (java.lang.String (roleName)	boolean is_caller_in_role(in string role);	
	public void setRollbackOnly() throws java.lang.IllegalState	void set_rollback_only() raises IllegalState;	
Interface	SessionContext extends EJBContext	TransientContext::CCMContext	
Operation	public EJBObject getEJBObject() throws java.lang.IllegalState	CORBA::Object get_CCM_Object() raises (IllegalState);	this will be the component reference
Interface	EntityContext extends EJBContext	PersistentContext::CCMContext	
Operation	public EJBObject getEJBObject() throws java.lang.IllegalState	CORBA::Object get_CCM_Object() raises (IllegalState);	this will be the component reference
	public java.lang.Object getPrimaryKey () throws java.lang.IllegalState	PrimaryKeyBase get_primary_key() raises (IllegalState);	

64.5.5 The Transaction Interfaces

Table 64-7 summarizes the transaction interfaces provided for bean-managed or component-managed transactions. Both EJB and CCM provide an accessor function in the context to obtain a reference to a transaction service. The transaction service supported for EJB is JTA, a subset of JTS which is equivalent to the CORBA transaction service (OTS). The transaction service supported for CORBA components is implemented by the component container as a wrapper over the CORBA transaction service. **Components::Transaction** is functionally equivalent to JTA (which is not a distinct compliance level for OTS) with the addition of **suspend** and **resume**.

Table 64-7 Comparing the EJB Transaction service (JTA) with CORBA component transactions

Construct	EJB Form	CCM Form	Notes
Module	javax.transaction	Components::Transaction	
Interface	UserTransaction	UserTransaction	
Operation	public void begin() throws NotSupported, SystemException	void begin () raises (NotSupported, SystemError);	SystemError to avoid confusion with System Exception

Table 64-7 Comparing the EJB Transaction service (JTA) with CORBA component transactions

Construct	EJB Form	CCM Form	Notes
	public void commit() throws RollbackException, HeuristicMixedException, HeuristicRollbackException, java.security.SecurityException, java.lang.IllegalStateException, SystemException	void commit() raises (Rollback, HeuristicMixed, HeuristicRollback, Security, IllegalState, SystemError	map CORBA system exceptions TRANSACTION_ROLLED_BACK to ROLLBACK and NO_IMPLEMENT to SECURITY
	public void rollback() throws java.security.SecurityException, java.lang.IllegalStateException, SystemException	void rollback() raises (Security, IllegalState, SystemError);	
	public void setRollbackOnly() throws SystemException	void set_rollback_only() raises (SystemError);	
	public int getStatus() throws SystemException;	Status get_status() raises (SystemError);	
	public void setTransactionTimeout (int seconds) throws SystemException	void set_transaction_timeout(in long to) raises (SystemError);	
		TranToken suspend() raises (NoTransaction, SystemError);	CCM supports suspend/resume which JTA does not
		void resume(in TranToken) raises (invalidToken, SystemError);	CCM supports suspend/resume which JTA does not

64.5.6 The Metadata Interfaces

The EJB component model supports a limited set of metadata through the **EJBMetaData** interface. The CORBA component model extends the CORBA interface repository to add component-unique metadata for components. This metadata is in addition to the metadata currently provided by the IR. When EJB clients access CORBA components, the container provider must provide an implementation of **EJBMetaData** which supports the necessary metadata from the Interface Repository or the component descriptors. This is described further in Chapter 8. When CORBA clients access EJB implementations, the Interface Repository is already populated for the **EJBHome** and **EJBObject** interfaces, enabling client requests to be satisfied. Table 64-8 compares the metadata supported by EJB and CORBA Components.

Issue – This table will be completed after the Interface Repository chapter is ready.

Table 64-8 Comparing component metadata between EJB and CORBA components

Construct	EJB Form	CCM Form	Notes
Module	javax.ejb	IR	
Interface	EJBMetaData	ComponentDef	
	public EJBHome getEJBHome()		
	public java.lang.Class getHomeInterfaceClass()		
	public java.lang.Class getRemoteInterfaceClass()		
	public java.lang.Class getPrimaryKeyClass()		
	public boolean isSession()		
	public boolean isStatelessSession()		

Component Container Architecture

66

This chapter describes the architecture of the **component container** as seen by the container provider.

Issue – It contains all new text taken from the CCM final submission

66.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Component Server”	66-200
“Containers Categories”	66-206
“Persistence Integration”	66-249
“Event Management Integration”	66-252

This chapter describes the architecture of the **component container** as seen by the container provider. The component container is a server-side framework built on the ORB, the Portable Object Adaptor (POA), and a set of CORBA services, which provides the runtime environment for a CORBA component. Component containers may be implemented by an existing ORB vendor or by companies not in that business today using the facilities of a CORBA 3.x ORB).

The container architecture in sections 66.1, and 66.2 of this chapter is described in terms of an exemplary design for building component containers on the POA using a **ServantLocator**. This is **not** the only possible design choice. Other designs are also possible although there are specific combinations of POA policies that cannot be made to work. These are indicated as rationale in the body of the text. A component container that exhibits the same behavior as the exemplary design presented in this chapter is conformant, even if it implements the container using a different design.

66.1 Component Server

A **component server** is a process which includes an arbitrary number of **component containers**:

- Each container has an associated **container API type**, which describes it's interaction with the component, and an associated **CORBA usage model**, which describes its interaction with the POA, the ORB, and a set of CORBA services.
- Each container supports a single container API type and manages a specific **component category**. Multiple component instances of the same component category can be deployed in the same container.
- Each container includes (or is associated with) a specialized POA¹ which is responsible for creating references and managing servants for the components in that container.
- A container is created by a **container manager**, which is a factory for component containers, based on descriptive information packaged with the component.
- Container managers themselves are created as part of the installation and deployment process for CORBA components. The details of deployment are described in Section 69.8, "Property File Descriptor," on page 69-321.
- A component container can be an EJB container by supporting one of the EJB container API types (Session Bean or Entity Bean). More information on integrating EJB containers with CORBA is provided in Chapter 64, "Integrating with Enterprise JavaBeans".

1. The term "POA" is used to refer to not only the interface **POA**, but all the related interfaces (**ServantManager**, **ServantLocator**, etc.) necessary to create references and activate object instances in response to client requests.

The overall architecture is depicted in Figure 66-1 below:

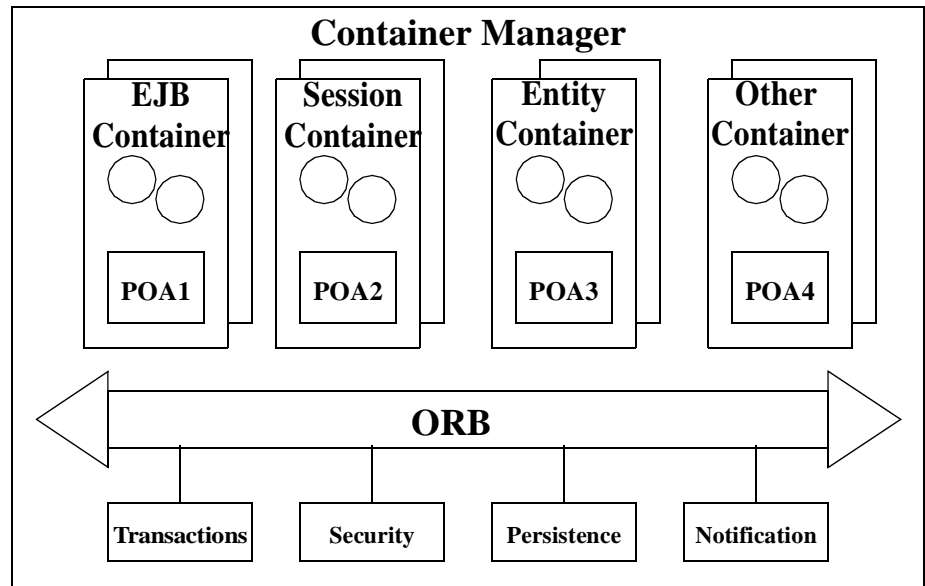


Figure 66-1 A Component Server

A component container is created as a result of component deployment as outlined in Section 69.8, “Property File Descriptor,” on page 69-321. The container specification is translated by the container manager into a set of POA policies, a container API type, and a set of CORBA service bindings that will be used by the container. This enables the container to implement **internal** interfaces, based on these bindings, which offer services to the component and invoke **callback** interfaces which the component developer must implement.

66.1.1 Component Levels

CORBA components define two levels of component functionality - basic and extended. These differ in the number of APIs and related functions made available to the component implementor. This distinction has no effect on the CORBA usage model or the **container API type** but does effect reference creation and which CORBA services are supported by the container. These distinctions are noted at the appropriate points in the text.

66.1.2 POA Creation

A POA is used to create references that will be exported to clients and to handle activation of component instances when operation requests are received. Creating a container usually involves the creation of a POA² for container use. The CORBA usage model associated with a particular container API type determines some of the policies which must be associated with the POA. These have been previously described in Section 62.2, “The Server Programming Environment. Others, which are orthogonal to

the container functionality (e.g. the use of firewall proxies), can be passed as input to the container manager. It is the responsibility of the container manager to then create a POA which satisfies these requirements.

CORBA::ORB::resolve_initial_references with an **ObjectID** of “RootPOA” is used to locate the root POA. The component category determines the CORBA usage model and some of the POA policies which must be used. The container manager uses this information to create a POA and its associated interfaces and to bind the API framework associated with the container API type.

*The container manager design for creating a POA described below uses a **ServantLocator** architecture which enables specialized **ServantManager** interfaces to implement the container function by being on the invocation path for all requests directed to the component. The API frameworks and their associated deployment descriptors defined for the **container API types** in this specification require the container to intervene before and after each operation request to implement the required function. This precludes certain POA policy choices, e.g. the use of a **ServantActivator** which is only called when the requested object is not in the POA's active object map. While other designs using different POA policies may be possible, this one was chosen because it best describes how the container behavior needs to be implemented.*

The steps required are as follows:

- The **CORBA::Policy** objects required by the POA are created with the proper values. The CORBA usage model requires or (in some cases) suggests specific POA policies. An example of a set that will work for each CORBA usage model can be found in Section 62.2, “The Server Programming Environment.
- A POA is created using the **POA::create_poa** operation specifying a sequence of the **Policy** objects created in the previous step as input. The complete set of **Policy** objects includes the mandatory set (dictated by the CORBA usage model), the orthogonal set (specified as input to container creation), and the implementation-specific set (chosen by the container provider to deliver the proper semantics).
- The container API type value is used to determine which **ServantManager** should be assigned to the POA (**POA::set_servant_manager**).

In the exemplary design, we use a unique **ServantManager** for each container API type (session and entity) defined by CORBA components. For EJB CORBA containers, there would also be two container API types corresponding to the EJB **Session Bean** and **Entity Bean**.

- The newly created POA is then activated (**POA::activate**)

In this exemplary design, a different container is defined for each component category and the container implementation is actually provided by the **ServantManager**. A **ServantLocator** design allows the container to be on the invocation path for every operation request. These component POAs specify the **USE_SERVANT_MANAGER**

2.It may be possible in some cases to actually use the root POA. This is not excluded, but has not been validated.

policy, enabling a **ServantManager** to be used to associate a servant with the request to instantiate the object. In standard CORBA, the **ServantManager** interface is implemented by user applications, but in the exemplary design for use with components, the specialized **ServantLocator** is implemented by the container provider.

There is a high degree of overlap between many of the component categories and their requirements for CORBA usage model making it feasible to build a single container that supports more than one component category. The exemplary design uses a container per component category for simplicity. Mapping a component implementation to a container by component category is a function of deployment and supports either a container per component category, as in the exemplary design, or multiple component categories in a single container as valid implementation choices.

66.1.3 Binding the Container to CORBA services

Basic CORBA components for all container API types defined in this specification use the following CORBA services:

- security
- transactions
- naming

Extended CORBA components for all container API types defined in this specification also support the following additional CORBA services:

- persistence
- notification

As part of container creation, accessibility to these CORBA services must be established and bindings created. At a minimum, this includes the use of the **resolve_initial_references** operation on **CORBA::ORB** to obtain initial references to these services. It also includes processing any container specific configuration data required for a particular service, e.g.

- setting up the channels to be used for emitting and consuming events,
- creating and initializing database connections to be used for persistence, and
- determining the naming context to be used to resolve component local names.

66.1.4 Container API Frameworks

The container API types defined by this specification provide **frameworks** into which a CORBA component is deployed. We define two container API types and their associated APIs in this specification. The EJB **SessionBean** and **EntityBean** interfaces represent two additional container API types. Each framework manages interactions with the ORB, the POA, and the CORBA services on behalf of the

CORBA component it supports, allowing the component developer to concentrate on application logic. The major functions handled by the API frameworks (in association with the ORB, POA, and the CORBA services) include:

- creating object references
- factories and finders
- transactions
- security
- events
- persistence

A brief description of each of these is provided in the following sections.

66.1.4.1 Creating Object References

In CORBA, object references are created and managed by the POA. A component container creates these reference with specialized information which comes from either the container provider, the component implementor, or the persistence provider, depending on both the component category and the deployment options specified.

For basic CORBA component containers and the EJB container API types, the container provider must manage object reference creation itself since these are not exposed to the programmer. The basic container is also responsible for binding references to the component home in the CORBA naming service so they can be accessed by the client as specified in the client programming model (Section 62.5, “The Client Programming Model). For EJB containers, only **EJBObject** and **EJBHome** have externally visible object references and these are implemented by the container, not the EJB programmer (see Section 64.3, “CORBA Component views for EJBs,” on page 64-175 for more details).

66.1.4.2 Factories and Finders

Factory and finder operations are declared using the **home** IDL declaration and are associated with the component’s home interface. All basic containers provides access to this interface at runtime. Extended containers also support a set of operations for externalizing component homes for use by external clients.

For EJB container API types, factories and finders are defined on **EJBHome** using a naming scheme defined by the Enterprise JavaBeans 1.1 specification.

66.1.4.3 Transactions

The container interacts with the CORBA transaction service on behalf of the component. Transaction policies, defined in the deployment descriptor, are translated into CORBA transaction service operations. For CORBA components with self-managed transactions, the container also provides the

Transaction::UserTransaction interface, a simplified form of the demarcation part of the CORBA transaction service which the component implementor uses to support transaction functions at runtime.

For EJB container API types, the **javax.jts.UserTransaction** interface (which is a subset of **Transaction::UserTransaction**), is mapped to the CORBA transaction service.

66.1.4.4 Security

The container relies on the CORBA security service to implement access control based on security policies defined in the deployment descriptor. The container also provides security operations which the component implementor uses to support security functions at runtime.

66.1.4.5 Events

Extended CORBA components shall have access to an event service supported by the container. The container provider is responsible for setting up and managing the event channels used by CORBA notification to support the component event model. The component event model relies on configuration information, local to the container implementor, to handle quality of service properties, filters, and the number and types of event channels. The container also provides access to the **Event** interface, which provides the mapping between the component event model and CORBA notification, to allow the component to both generate and process events. Integrating the component event model with CORBA notification is addressed in Section 66.4.

At the time this specification was produced, the EJB container API types did not support an event API although the Java Messaging Service (JMS) API has been defined separately (from EJB) and supports similar function. An event API is targeted for EJB 2.0.

66.1.4.6 Persistence

For extended components, persistence is supported by containers for the entity container API type. Component containers supporting the session container API type do not support persistence. Component containers for basic components do not offer a persistence API. For extended components, the container provides access to a set of APIs provided by the CORBA persistent state service which offers the functions necessary to implement self-managed persistence. Persistence considerations are covered in more detail in Section 66.3.

For basic components, all entity container API types (including EJB Entity Beans) support a **getPrimaryKey** operation on the context equivalent to the **get_primary_key** operation on component homes which declare a primary key. Component persistence (both container-managed and self-managed) is assumed to be implemented using JDBC or some other unspecified persistence API (e.g. JSQL or ODBC) and is therefore not defined as part of these container API types.

66.1.4.7 Threading

CORBA components support two forms of thread safety: **serialize**, and **multithread**. These choices are described in Section 69.4.5.54, “The threading Element,” on page 69-295. The container implements these choices by either ensuring that only a single thread enters a component at a time (**serialize**) or by allowing multiple threads to enter a component simultaneously (**multithread**).

Basic container API types (including EJB) support only the **serialize** threading policy.

66.2 Containers Categories

The exemplary design delineates **container categories** corresponding to the four component categories with their associated container API types, two **container categories** for the EJB container API types, and an empty **container category** to support creation of user-defined frameworks:

- The **Service** container which manages the service component designed for high-performance access to stateless CORBA components (Section 66.2.2).
- The **Session** container which manages the session component for stateful CORBA components with transient state (Section 66.2.3).
- The **Process** container which manages stateful process components which encapsulates all data access in the server using any persistence mechanism (Section 66.2.4).
- The **Entity** container which manages stateful entity components which shares data access responsibility between the client and the server using any persistence mechanism (Section 66.2.5).
- The **EJBSession** container which manages EJB **Session Beans** (Section 66.2.6).
- The **EJBEntity** container which manages EJB **Entity Beans** (Section 66.2.7).
- The **Empty** container which makes the entire suite of CORBA interfaces available to a component’s implementation without restriction (Section 66.2.1).

These container categories are one to one with their component categories. The relationship between component categories, container API types and CORBA usage models was described previously in Section 62.2, “The Server Programming Environment. The following sections describe each of the container categories in more detail.

66.2.1 The Empty Container

The **Empty** container exposes all CORBA functions directly to the component developer. No framework is provided to simplify programming, however all the functions necessary to build such a framework are available. The component developer can choose any function currently defined in CORBA. The empty container is the means by which the advanced functions of CORBA components (e.g. multiple

interfaces, packaging, and deployment) are made available to any CORBA applications, including those that do not fit the profiles of the other component categories. This is illustrated in Figure 66-1 below:

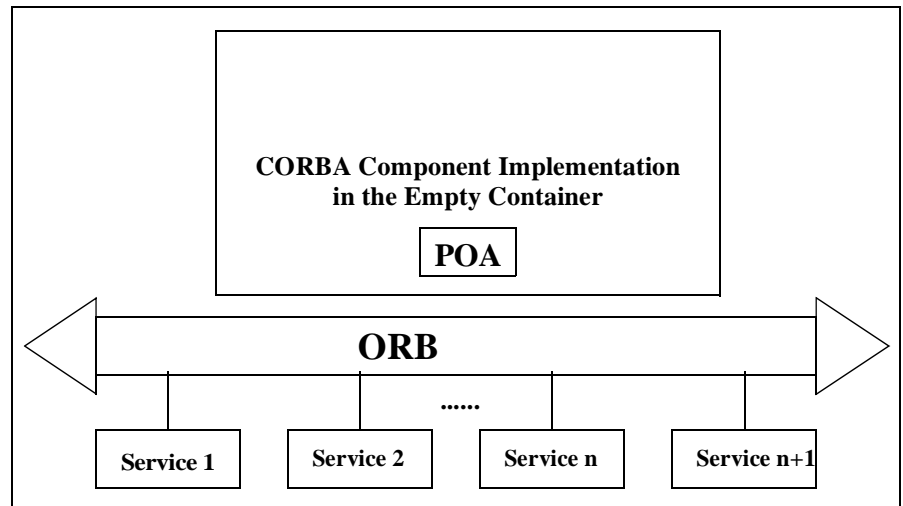


Figure 66-1 The Empty Container

Note that a CORBA component deployed in the empty container can use any arbitrary set (including the null set) of CORBA services. This specification places no constraints on what can be used within the existing CORBA architecture.

66.2.2 The Service Container

The **service** container implements the runtime environment for a service component. A service container can be implemented using a POA with the policies enumerated in Table 66-1. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-1 POA Policies for a Service Container

Policy Name	Required Value	Design Value
Thread	ORB_CTRL_MODEL	
Lifespan	TRANSIENT	
Object Id Uniqueness		N/A
Id Assignment		SYSTEM_ID
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN

Table 66-1 POA Policies for a Service Container

Policy Name	Required Value	Design Value
Transaction Policy	ALLOWS_SHARED	
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL_MODEL** is required to allow the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

A lifespan policy value of **TRANSIENT** is required since service components have neither state nor identity³.

Object Id uniqueness

The Object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN** as it is in the exemplary design.

Id assignment

An Id assignment policy value of **SYSTEM_ID** allows the POA to assign **Objectld** values. Since service components have no identity, the service container has no need to manage **Objectld** assignment.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator** in the exemplary design.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

³In practice, the distinction between PERSISTENT and TRANSIENT references is difficult, if not impossible, to observe. The semantics associated with the definition of TRANSIENT are closer to the semantics of this category of component.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the servant manager.

66.2.2.1 *Creating Object References*

For **service** components, **ObjectIds** have no meaning since a **service** component has neither state or identity. The exemplary design allows the POA to create them transparently to both the container and the component.

66.2.2.2 *Factories and Instances*

A component home implementation for a **service** component creates object references and component instances in response to the client's **create** requests. Extended **service** components may register their home with the **HomeFinder** to make it available to clients through find operations or the component home can be bound in the name service. For **service** components, the component instance and its home need not be collocated. Since instances have no state, they can be created anywhere when a request is received. Object references for both the component's supported interfaces and any provided interface are created by the POA within the service container.

66.2.2.3 Invoking an Operation

Figure 66-1 below outlines the steps necessary to make an operation invocation on a service component:

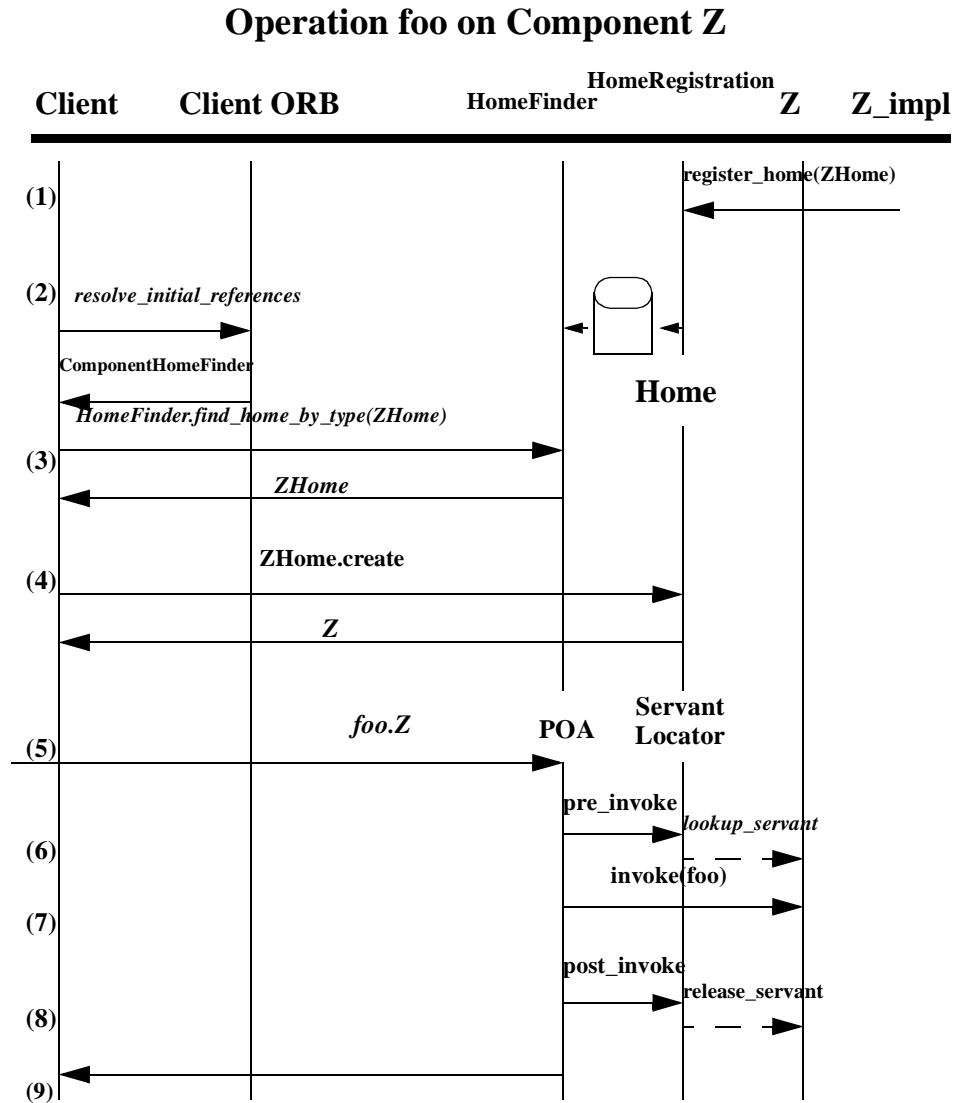


Figure 66-1 Using a Service Component

1. Component implementation registers a **service** component factory (i.e. its home) with the **HomeFinder** (**HomeRegistration.register_home**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.

3. Client uses the **HomeFinder.find_home_by_type** operation to find a component home (**Zhome**) that creates component instances of type **Z**.
4. Client invokes a **create** operation on the component home (**ZHome.create**). Since **Z** is a **service** component, the home creates a reference and defers activation.
5. Client invokes the **foo** operation on **Z** (**Z.foo**).
6. The POA invokes the **ServantLocator** and requests an **executor** to process the request (**ServantLocator.pre_invoke**). The **ServantLocator** locates an appropriate **executor** or creates a new one. It returns the associated servant to the POA.
7. The POA dispatches the request to the component implementation (**Invoke Z.foo**).
8. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**). The **ServantLocator** releases the associated **executor** to the pool.
9. The POA returns **foo** response to the client.

66.2.2.4 Servant Lifetime Management

The service component requires a servant lifetime policy of **method**. A servant with a **method** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface and passivated in the **post_invoke** following the operation invocation. This behavior is shown in Figure 66-1 below:

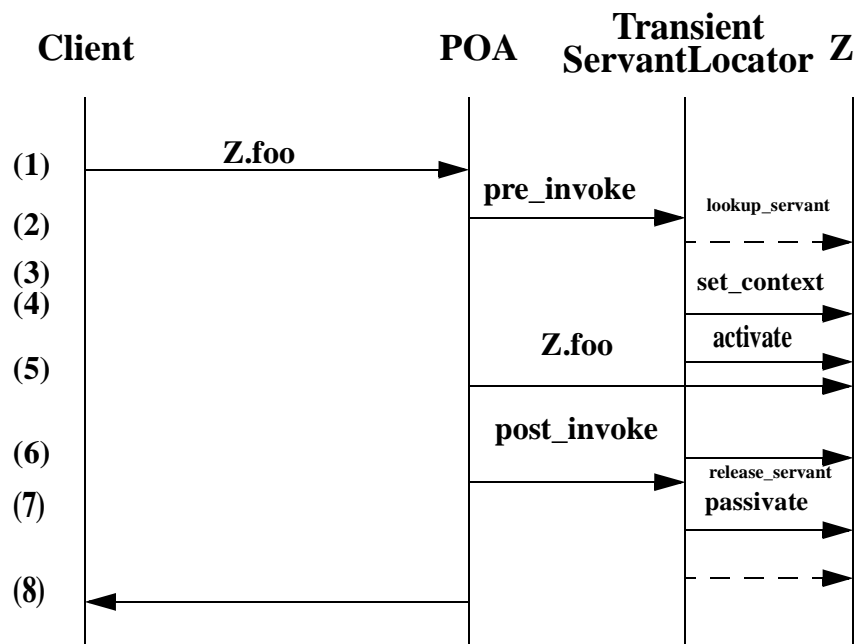


Figure 66-1 Service Container with a Method Lifetime Policy

1. Client invokes **foo** operation on **Z** (**Z.foo**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
4. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
5. POA then dispatches **foo** operation to **Z**.
6. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
7. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement the **passivate** operation.
8. POA then returns **foo** response back to client. Since the servant lifetime policy is **method**, the **executor** is released.

66.2.3 The Session Container

The **session** container implements the runtime environment for a session component. A session container can be implemented using a POA with the policies enumerated in Table 66-2. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-2 POA Policies for a Session Container

Policy Name	Required Value	Design Value
Thread	ORB_CTRL_MODEL	
Lifespan	TRANSIENT	
Object Id Uniqueness		N/A
Id Assignment	USER_ID	
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN
Transaction Policy	ALLOWS_SHARED	
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL_MODEL** is required to allow the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

A lifespan policy value of **TRANSIENT** is required since session components have transient state and identity⁴.

Object Id uniqueness

The Object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN**.

Id assignment

An Id assignment policy value of **USER_ID** is required to allow the **session** container to assign unique **ObjectIds** with input from the component. This supports a structuring of **ObjectId** values which the container can exploit within its implementation.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the servant manager.

4. In practice, the distinction between **PERSISTENT** and **TRANSIENT** references is difficult, if not impossible, to observe. The semantics associated with the definition of **TRANSIENT** are closer to the semantics of this category of component.

66.2.3.1 *Creating Object References*

For basic session components, **ObjectIds** are always managed by the session container without involvement from the component implementor. Extended session components create their own references using container APIs. The container implementor is responsible for maintaining uniqueness. This permits **ObjectIds** to be encapsulated by the container provider in implementation specific ways.

66.2.3.2 *Factories and Instances*

The home implementation for a **session** component creates object references and component instances in response to the client's **create** requests. Extended **session** components may register their home with the **HomeFinder** to make it available to clients through find operations or the component home can be bound in the name service. For **session** components, the component instance and the factory must be colocated. Object references for both the component's supported interfaces and any provided interface are created by the POA within the session container.

66.2.3.3 Invoking an Operation

Figure 66-1 below outlines the steps necessary to make an operation invocation on a session component:

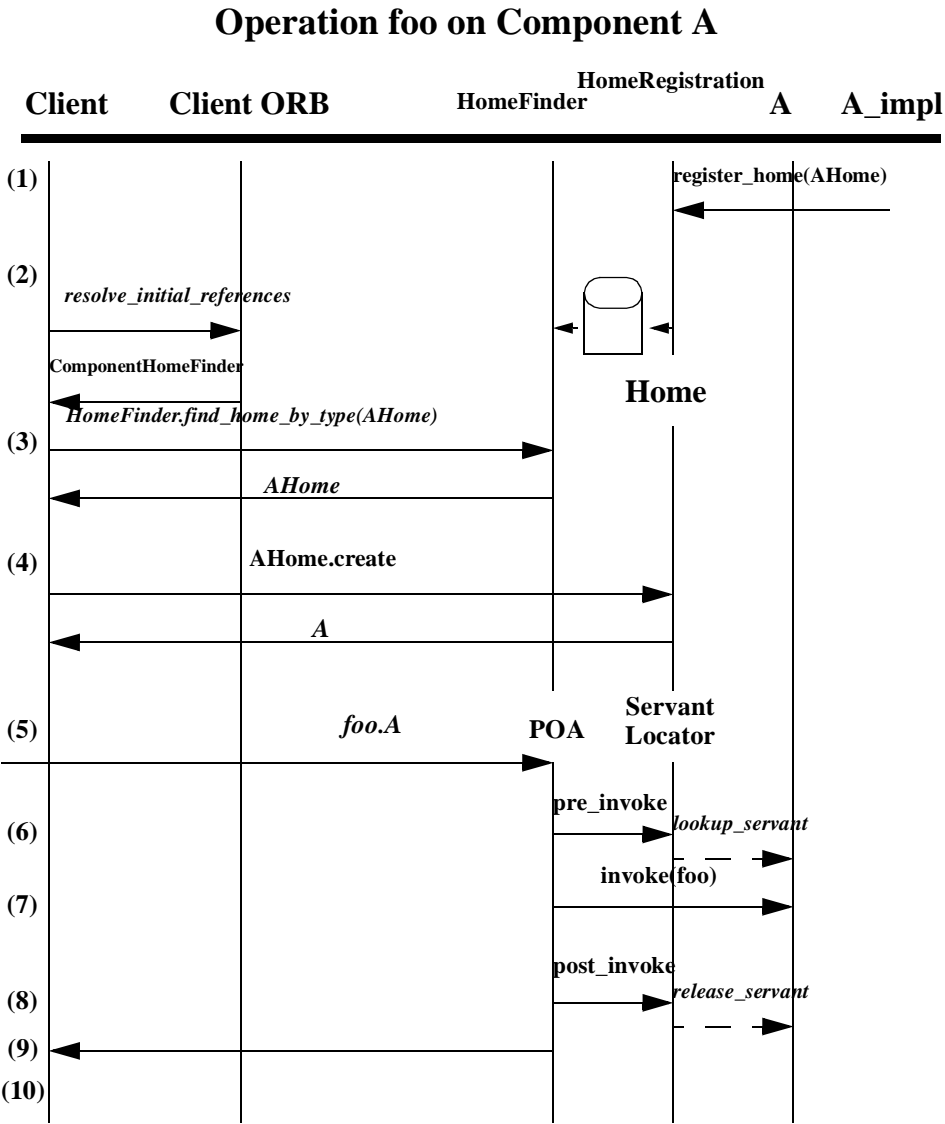


Figure 66-1 Using a Session Component

1. Component implementation registers a **session** component's home with the **HomeFinder** (**HomeRegistration.register_home**).
2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.

3. Client uses the **HomeFinder.find_home_by_type** operation to find a component home (**Ahome**) that creates component instances of type **A**.
4. Client invokes a **create** operation on the component home (**AHome.create**). Since **A** is a **session** component, the home creates a reference and may defer activation until the first operation invocation.
5. Client invokes the **foo** operation on **A** (**A.foo**).
6. The POA invokes the **ServantLocator** and requests an **executor** to process the request (**ServantLocator.pre_invoke**). The **ServantLocator** locates an appropriate **executor** or creates a new one. The POA dispatches the request to the component implementation (**Invoke A.foo**).
7. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
8. POA then returns **foo** response back to client.
9. Steps [5] through [8] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor** to the pool.

66.2.3.4 *Servant Lifetime Management*

The session container supports multiple servant lifetime policy values. An executor is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface and is passivated in the **post_invoke** following the expiration of the servant lifetime policy. This is illustrated in the following sections:

Method Lifetime

A session component with a **method** lifetime policy has its **executor** activated on every **pre_invoke** and passivated on every **post_invoke**. This behavior is shown in Figure 66-1:

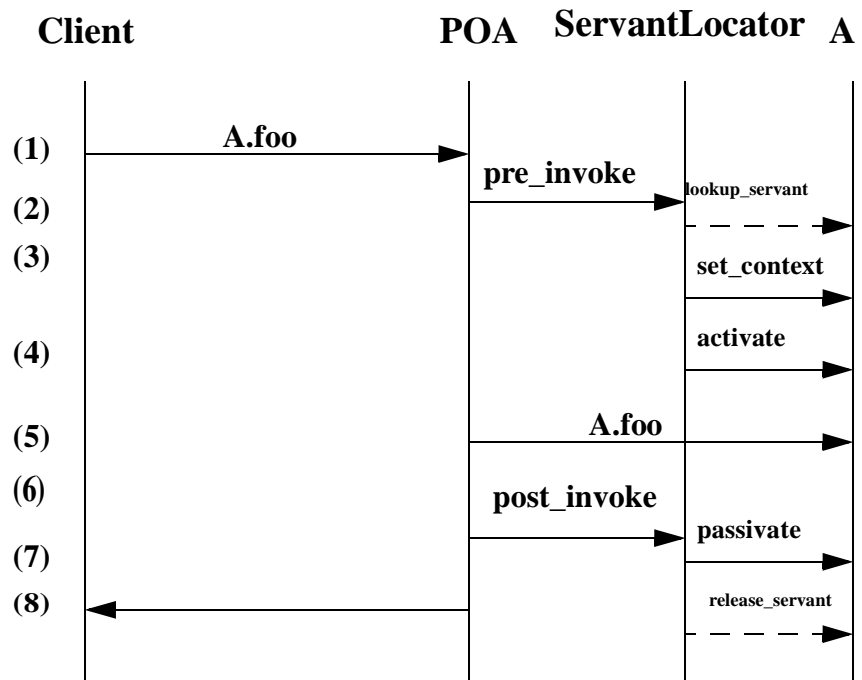


Figure 66-1 Session component with a Method Lifetime Policy

1. Client invokes **foo** operation on **A** (**A.foo**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
4. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
5. POA then dispatches **foo** operation to **A**.
6. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
7. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement the **passivate** operation.
8. POA then returns **foo** response back to client and releases **executor**.

Transaction Lifetime

A session component with a **transaction** lifetime policy is activated on the first **pre_invoke** within a new transaction. Subsequent **pre_invoke** operations do not cause activation. Passivation occurs when the current transaction completes (successfully or unsuccessfully). The **ServantLocator** implements this policy using the CORBA transaction service **CosTransactions::Synchronization** interface. This behavior is shown in Figure 66-2:

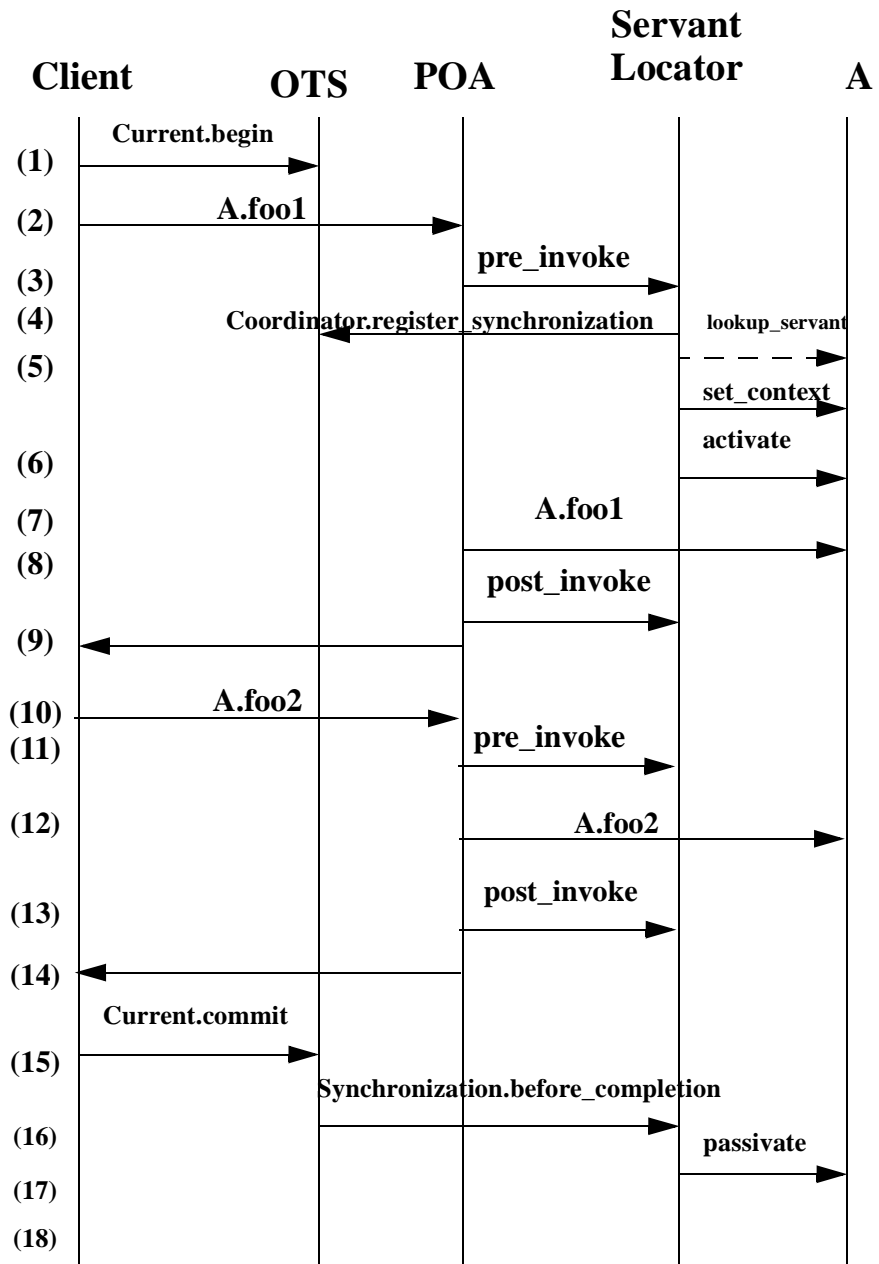


Figure 66-2 Session Component with a Transaction Lifetime Policy

1. Client begins a transaction with the CORBA transaction service (**Current.begin**)
2. Client invokes **foo1** operation on **A** (**A.foo1**).
3. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
4. **ServantLocator** registers a **Synchronization** object with the CORBA transaction service (**Coordinator.register_synchronization**) to be called by the CORBA transaction service at the start of the commit process.
5. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
6. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
7. POA then dispatches **foo1** operation to **A**.
8. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
9. POA then returns **foo1** response back to client.
10. Client invokes **foo2** operation on **A**.
11. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**). Since **A** is already active, the **ServantLocator** returns to the POA.
12. POA then dispatches **foo2** operation to **A**.
13. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
14. POA then returns **foo2** response back to client.
15. Client attempts to terminate the transaction by calling commit (**Current.commit**)
16. CORBA transaction service notifies **ServantLocator** prior to the start of phase one of commit (**Synchronization.before_completion**).
17. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement the **passivate** operation.
18. CORBA transaction service continues the two-phase commit process.

Component Lifetime

A session component with a **component** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application requested passivation or when the process terminates, whichever occurs first. This behavior is shown in Figure 66-3 on page 220.

Container Lifetime

A session component with a **container** lifetime policy is activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application-requested passivation or in the **post_invoke** following an operation when the system needs to reclaim the memory, whichever occurs first. This behavior is identical to **component** behavior, except that failures can be simulated when the container determines that it needs to reclaim the memory associated with this component making it more likely that the final response will be returned to the client. This behavior is captured in Figure 66-3 below.

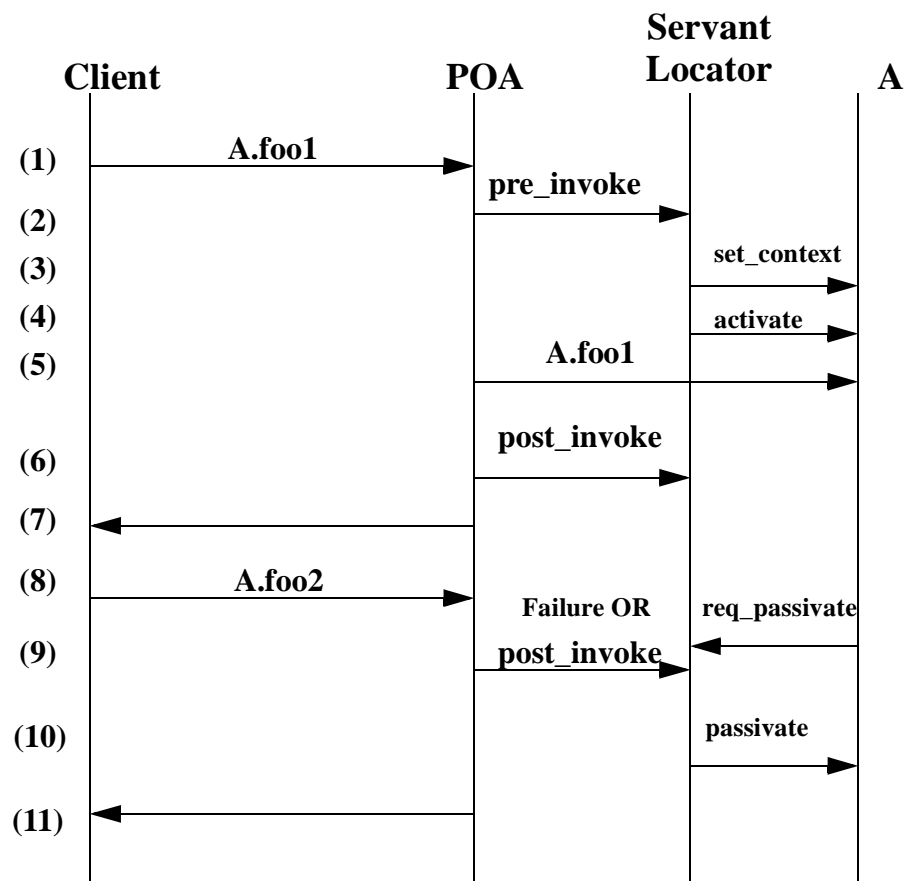


Figure 66-3 A Session Component with Component or Container Lifetime Policy

1. Client invokes **foo1** operation on **A** (**A.foo1**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
4. **ServantLocator** then invokes **activate** callback operation. The component developer must implement the **activate** operation.
5. POA then dispatches **foo1** operation to **A**.
6. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**). Since activation policy is **component** or **container**, the **ServantLocator** just returns to the POA.
7. POA then returns **foo1** response back to client.
8. Client continues invoking **foo2** operation (**A.foo2**). Either a failure occurs or **A** requests to be passivated (**Session2Context.req_passivate**).
9. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
10. **ServantLocator** then invokes **passivate** callback operation. The component developer must implement the **passivate** operation.
11. POA then returns **foo2** response back to client (if possible).

66.2.4 The Process Container

The **process** container implements the runtime environment for a process component. A process container can be implemented using a POA with the policies outline in Table 66-3. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-3 POA Policies for a Process Container

Policy Name	Required Value	Design Value
Thread	ORB_CTRL_MODEL	
Lifespan	PERSISTENT	
Object Id Uniqueness		N/A
Id Assignment	USER_ID	
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN
Transaction Policy	ALLOWS_SHARED	

Table 66-3 POA Policies for a Process Container

Policy Name	Required Value	Design Value
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL__MODEL** is required to allow the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

A lifespan policy value of **PERSISTENT** is required since process components have both persistent state and identity.

Object Id uniqueness

The object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN**.

Id assignment

An Id assignment policy value of **USER_ID** is required to allow the process container to assign unique **ObjectIds** with input from the component implementation and the persistence mechanism. This not only supports a structuring of **ObjectId** values which the container can exploit within its implementation, but also makes it possible for the component implementor or the persistence mechanism to locate state from the **ObjectId**.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the servant manager.

66.2.4.1 *Creating Object References*

The **process** container is responsible for creating and managing unique **ObjectIds** which can be used to locate an external copy of the component's persistent state. That state can be explicitly declared and managed by the container (**container-managed persistence**) or not declared and managed by the application (**self-managed persistence**). These **ObjectIds** are opaque both to the client and to the container, and may or may not use the CORBA persistence mechanism. This makes it possible to have factories for **process** components which create only object references and defer instance creation until an operation request is actually received. This enables workload to be distributed among several functionally equivalent servers.

66.2.4.2 *Factories and Instances*

The **process** component's home is responsible for creating references and exporting them to clients. Component instances are created on demand when a reference is used to invoke an operation.

Factory operations are typically invoked by clients but may also be invoked as part of the implementation of a specific interface provided by the component. A CORBA component implementation locates its home (which supports the factory operations) using the context provided by its container. Object references for both the component's interfaces and any provided interface are created by the POA which supports the container for that component.

66.2.4.3 Invoking an Operation

Figure 66-1 outlines the steps necessary to make an operation request on a process component:

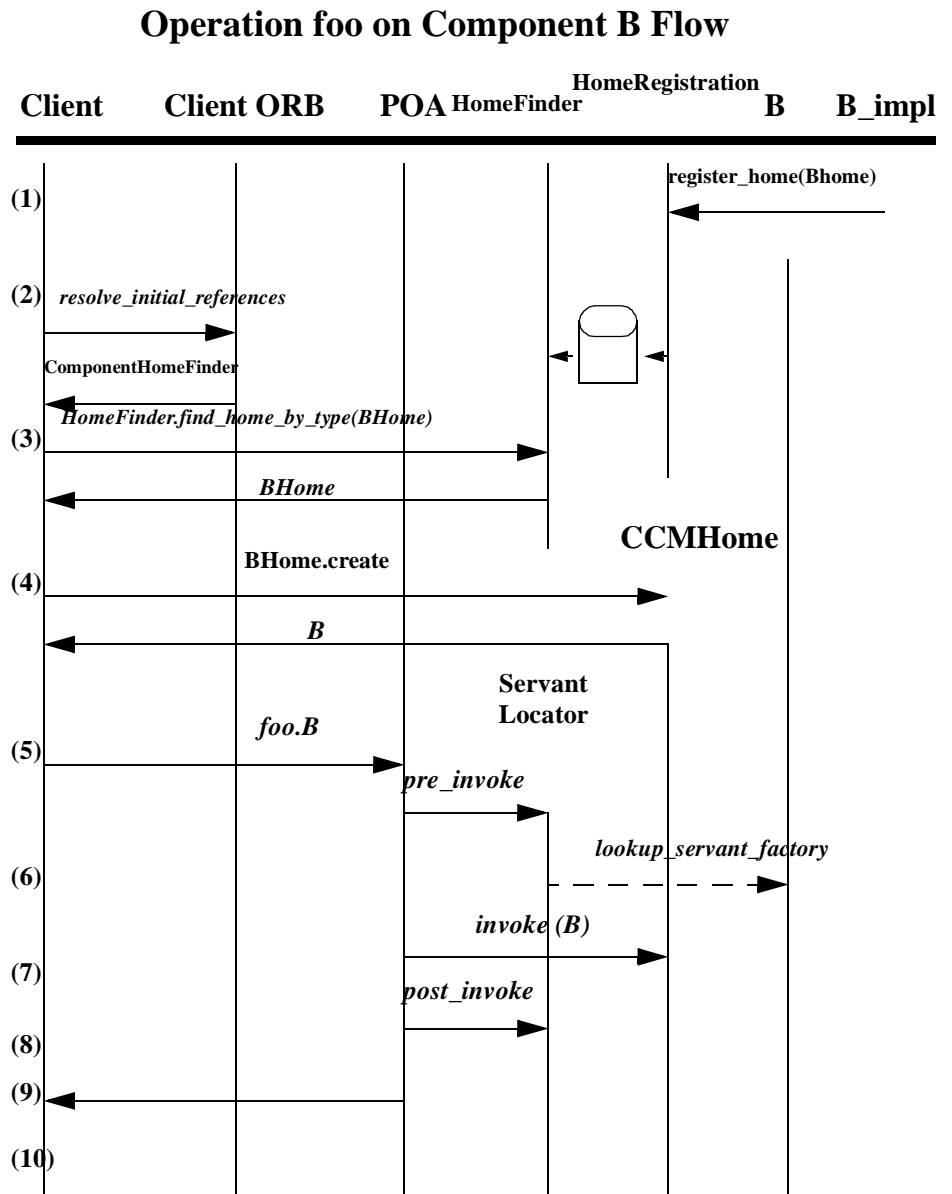


Figure 66-1 Using the Process Container

1. Component implementation registers a **process** component home with the **HomeFinder** (`HomeRegistration.register_factory`).

2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.
3. Client uses the **HomeFinder.find_home_by_type** operation to find a component home (**BHome**) that creates component instances of type **B**.
4. Client invokes a **create** operation on the component home (**BHome.create**). Since **B** is **process** component, the home need only create a reference; instance creation can be deferred until an operation is requested.
5. Client invokes the **foo** operation on **B** (**B.foo**). Since **B** is not active, the POA invokes the **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
6. The **ServantLocator** creates a new **executor** to handle the request. It then returns the associated servant to the POA to process the request.
7. The POA then dispatches the request to the servant (**invoke(B)**)
8. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
9. The POA returns **foo** response to client.
10. Steps [5] through [9] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor** to the pool.

66.2.4.4 *Servant Lifetime Management*

The process component can have multiple servant lifetime policies specified in its deployment descriptor. The **ServantLocator** implements these different policies by making activation decisions during **pre_invoke** and passivation decisions during **post_invoke**. This is illustrated in the following sections:

Method Lifetime

A process component with a **method** lifetime policy has its **executor** activated on every **pre_invoke** and passivated on every **post_invoke**. This behavior is shown in Figure 66-1:

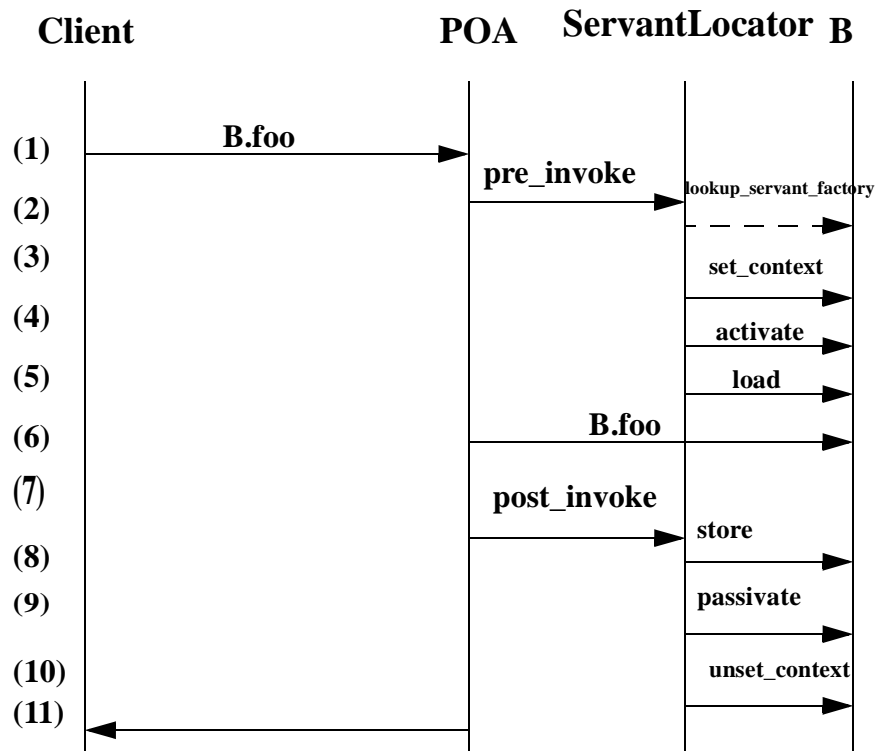


Figure 66-1 A Process Component with a Method Lifetime Policy

1. Client invokes **foo** operation on **B** (**B.foo**).
2. POA invokes **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
4. **ServantLocator** creates a new **B** and invokes the **activate** callback operation. For most component implementations, no action is required.
5. **ServantLocator** then invokes the **load** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **load** operation.
6. POA then dispatches **foo** operation to **B**.

7. When **foo** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
8. **ServantLocator** then invokes the **store** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement the **store** operation.
9. **ServantLocator** then invokes **passivate** callback operation. For most component implementations, no action is required.
10. **ServantLocator** invokes callback operation to unset the context and releases the **executor**.
11. POA then returns **foo** response back to client.

Transaction Lifetime

A process component with a **transaction** lifetime policy has its **executor** activated on the first **pre_invoke** within a new transaction. Subsequent **pre_invoke** operations do not cause activation. Passivation occurs when the current transaction completes

(successfully or unsuccessfully). The **ServantLocator** implements this policy using the CORBA transaction service **CosTransactions::Synchronization** interface. This behavior is shown in Figure 66-2:

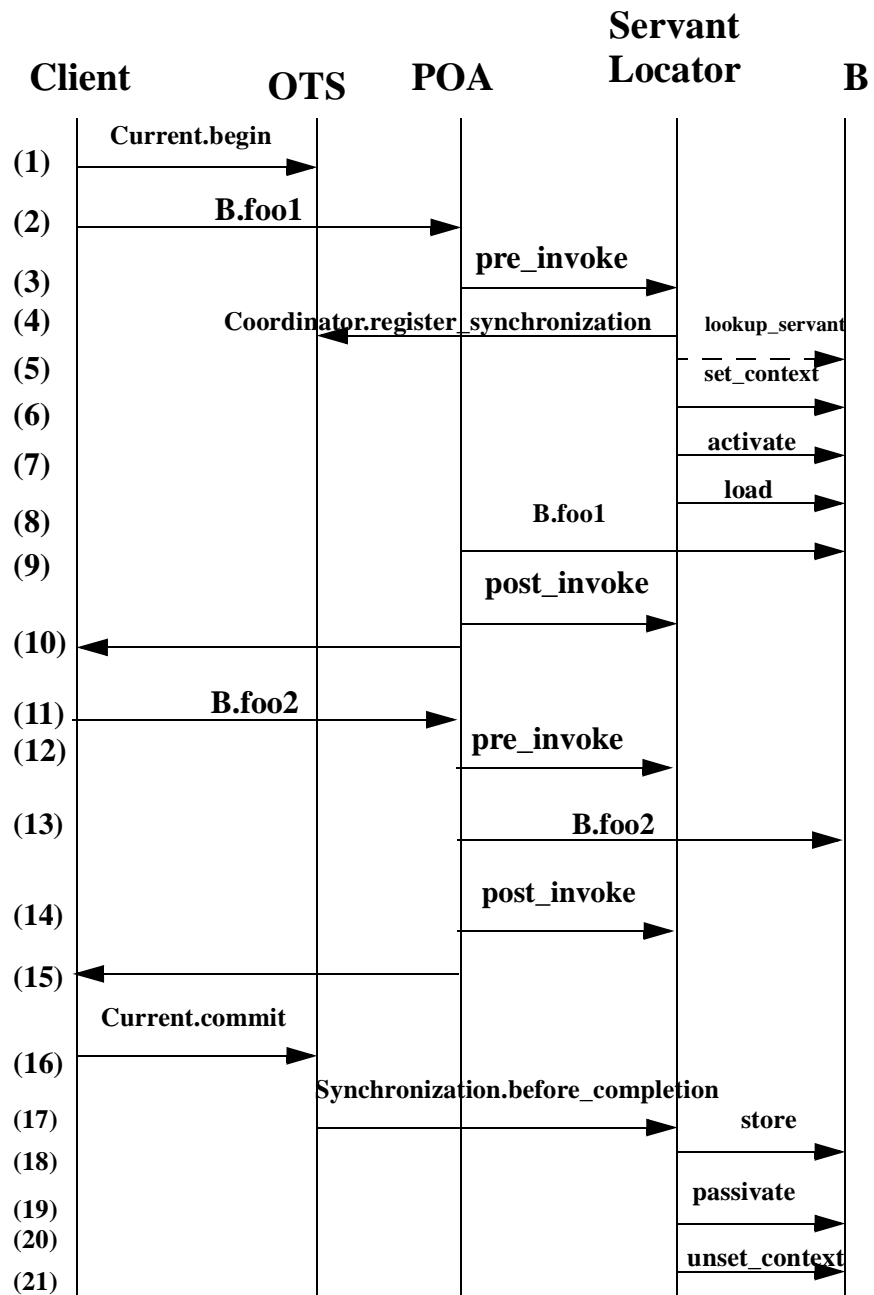


Figure 66-2 A Process Component with a Transaction Lifetime Policy

1. Client begins a transaction with the CORBA transaction service (**Current.begin**)

2. Client invokes **foo1** operation on **B** (**B.foo1**).
3. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
4. **ServantLocator** registers a **Synchronization** object with the CORBA transaction service (**Coordinator.register_synchronization**) to be called by the CORBA transaction service at the start of the commit process.
5. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
6. **ServantLocator** creates a new **B** and invokes **activate** callback operation. For most component implementations, no action is required.
7. **ServantLocator** then invokes **load** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **load** operation.
8. POA then dispatches **foo1** operation to **B**.
9. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
10. POA then returns **foo1** response back to client.
11. Client invokes **foo2** operation on **B**.
12. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**). Since **B** is already active, the **ServantLocator** returns to the POA.
13. POA then dispatches **foo2** operation to **B**.
14. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
15. POA then returns **foo2** response back to client.
16. Client attempts to terminate the transaction by calling **commit** (**Current.commit**).
17. CORBA transaction service notifies **ServantLocator** prior to the start of phase one of commit (**Synchronization.before_completion**).
18. **ServantLocator** then invokes **store** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement the **store** operation.
19. **ServantLocator** then invokes **passivate** callback operation. For most component implementations, no action is required.

20. **ServantLocator** invokes callback operation to unset context and releases the **executor**.
21. CORBA transaction service continues the two-phase commit process.

Component Lifetime

A process component with a **component** lifetime policy has its **executor** activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application requested passivation or when the process terminates, whichever occurs first. This behavior is shown in Figure 66-3 below.

Container Lifetime

A process component with a **container** lifetime policy has its **executor** activated on the first **pre_invoke** prior to an operation being dispatched on the component's interface. Passivation occurs either in the **post_invoke** following an application-requested passivation or in the **post_invoke** following an operation when the system needs to reclaim the memory, whichever occurs first. This behavior is identical to **component** behavior, except that failures can be simulated when the container determines that it

needs to reclaim the memory associated with this component making it more likely that the final response will be returned to the client. This behavior is captured in Figure 66-3 below.

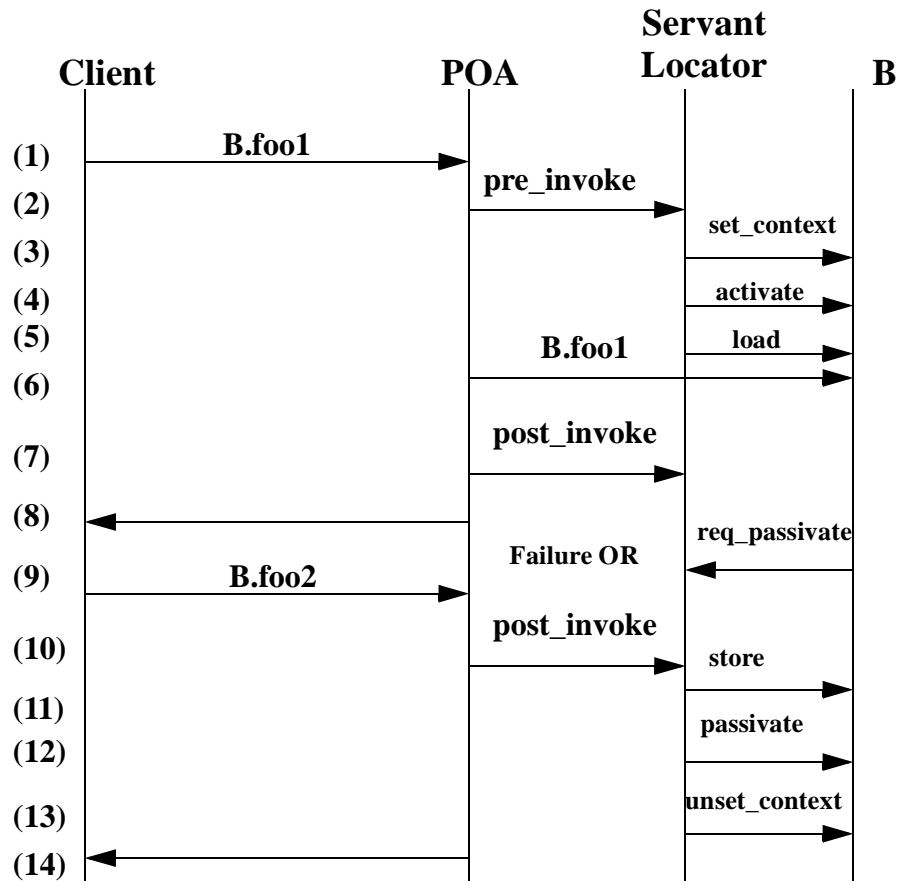


Figure 66-3 Process Component with Component or Container Lifetime Policies

1. Client invokes **foo1** operation on **B** (**B.foo1**).
2. POA invokes **pre_invoke** operation on **ServantLocator** (**ServantLocator.pre_invoke**).
3. **ServantLocator** finds an available **executor** and returns associated servant to the POA, and invokes callback operation to set context.
4. **ServantLocator** invokes **activate** callback operation. For most component implementations, no action is required.
5. **ServantLocator** then invokes **load** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the component developer must implement the **load** operation.

6. POA then dispatches **foo1** operation to **B**.
7. When **foo1** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**). Since activation policy is **component** or **container**, the **ServantLocator** just returns to the POA.
8. POA then returns **foo1** response back to client.
9. Client invokes **foo2** operation on **B** (**B.foo2**). Either a failure occurs or **B** requests to be passivated (**Entity2Context.req_passivate**).
10. When **foo2** operation completes, POA invokes **post_invoke** operation on **ServantLocator** (**ServantLocator.post_invoke**).
11. **ServantLocator** then invokes **store** callback operation. If the component has declared its abstract state using CORBA persistence, this callback will be executed as generated code. If no abstract state is declared, the generated code simply returns. If abstract state is declared not using CORBA persistence, the developer must implement the **store** operation.
12. **ServantLocator** then invokes **passivate** callback operation. For most component implementations, no action is required.
13. **ServantLocator** invokes callback operation to unset context and releases the **executor**.
14. POA then returns **foo** response back to client (if possible).

66.2.5 The Entity Container

The entity container provides the runtime environment for the entity component. A entity container can be implemented using a POA with the policies outlined in Table 66-4. These values are equivalent to those specified for the process container in Section 66.2.4. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-4 POA Policies for the Entity Container

Policy Name	Required Value	Design Value
Thread	ORB_CTRL_MODEL	
Lifespan	PERSISTENT	
Object Id Uniqueness		N/A
Id Assignment	USER_ID	
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN
Transaction Policy	ALLOWS_SHARED	

Table 66-4 POA Policies for the Entity Container

Policy Name	Required Value	Design Value
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL__MODEL** is required to allow the container to serialize access to components that are not thread safe (**serialize**). Thread safe components (**multithread**) will not be protected from multiple threads entering the component simultaneously.

Lifespan

A lifespan policy value of **PERSISTENT** is required since entity components have both persistent state and identity.

Object Id uniqueness

The object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN**.

Id assignment

An Id assignment policy value of **USER_ID** is required to allow the entity container to assign unique **ObjectIds** with input from the component implementation and the persistence mechanism. This not only supports a structuring of **ObjectId** values which the container can exploit within its implementation, but also makes it possible for the component implementor or the persistence mechanism to locate state from the **ObjectId**.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the servant manager.

66.2.5.1 *Creating Object References*

The **entity** container is responsible for creating and managing unique **ObjectIds** which can be used to locate an external copy of the component's persistent state. That state can be explicitly declared and managed by the container (**container-managed persistence**) or not declared and managed by the application (**self-managed persistence**). The entity container supports operations for associating primary keys with a **ComponentId** (**cid**). Every entity component instance is associated with one and only one primary key. The entity container provides operations on its **ServantLocator** to create an **ObjectId** from a **cid**.

66.2.5.2 *Factories and New Instances*

A **entity** component's home is responsible for both creating references and creating new instances of entity components. Since entity components are also incarnations in a persistent store, creating a new instance of the entity component has the effect of creating a new record in a persistent store.

Factory operations are typically invoked by clients but may also be invoked as part of the implementation of a specific interface provided by the component. The entity component implementation locates its home (which supports the factory operations) using the context provided by its container. Object references for both the component's interfaces and any provided interface are created by the POA which supports the container for the entity component.

66.2.5.3 Invoking an Operation on a New Instance

Figure 66-1 shows the necessary steps to make an operation request on a new entity component:

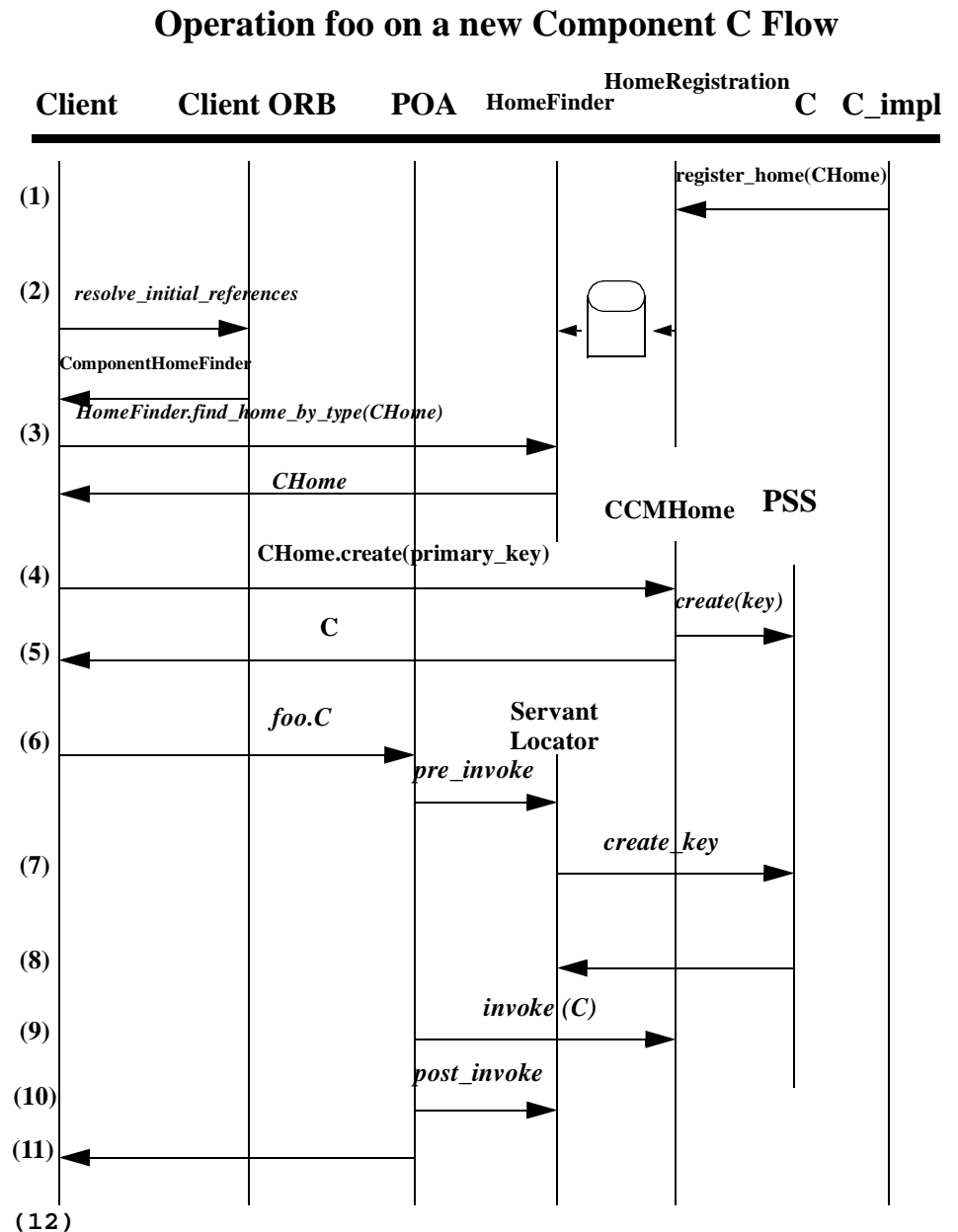


Figure 66-1 Using the Entity Container to Create new Entity Components

1. Component implementation registers the **entity** component home with the **HomeFinder** (**HomeRegistration.register_factory**).

2. Client uses **ORB.resolve_initial_references** to get a reference to the **ComponentHomeFinder**. Since the **HomeFinder** is a righteous CORBA object, it's implementation may be located anywhere.
3. Client uses the **HomeFinder.find_home_by_type** operation to find a component home (**CHome**) that creates component instances of type **C**.
4. Client invokes a **create** operation on the component home (**CHome.create**) using a primary key. Since **C** is an **entity** component, the home must talk to a persistence mechanism to create a new record in the persistent store using the same primary key.
5. A reference to **C** is returned to the client.
6. Client invokes the **foo** operation on **C** (**C.foo**). Since **C** is not active, the POA invokes the **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
7. The **ServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism finds the appropriate incarnation and returns it to the **ServantLocator**.
8. The **ServantLocator** creates a new **executor** to handle the request. The associated servant is returned to the POA to process the request.
9. The POA then dispatches the request to the servant (**invoke(C)**)
10. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
11. The POA returns **foo** response to client.
12. Steps [6] through [11] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor**.

66.2.5.4 Finders and Existing Instances

The **entity** component may also correspond to an existing element in a persistent store. If so, a finder is responsible for locating the **PersistentId** and associating an incarnation with an instance of the entity component. The home interface for entity components supports finder operations.

The client will use either the **HomeFinder** or the Naming service to locate the home interface. A CORBA component implementation can locate its home interface using the context provided by its container.

66.2.5.5 Invoking an Operation on an Existing Instance

Figure 66-1 shows the necessary steps to make an operation request on an existing entity component:

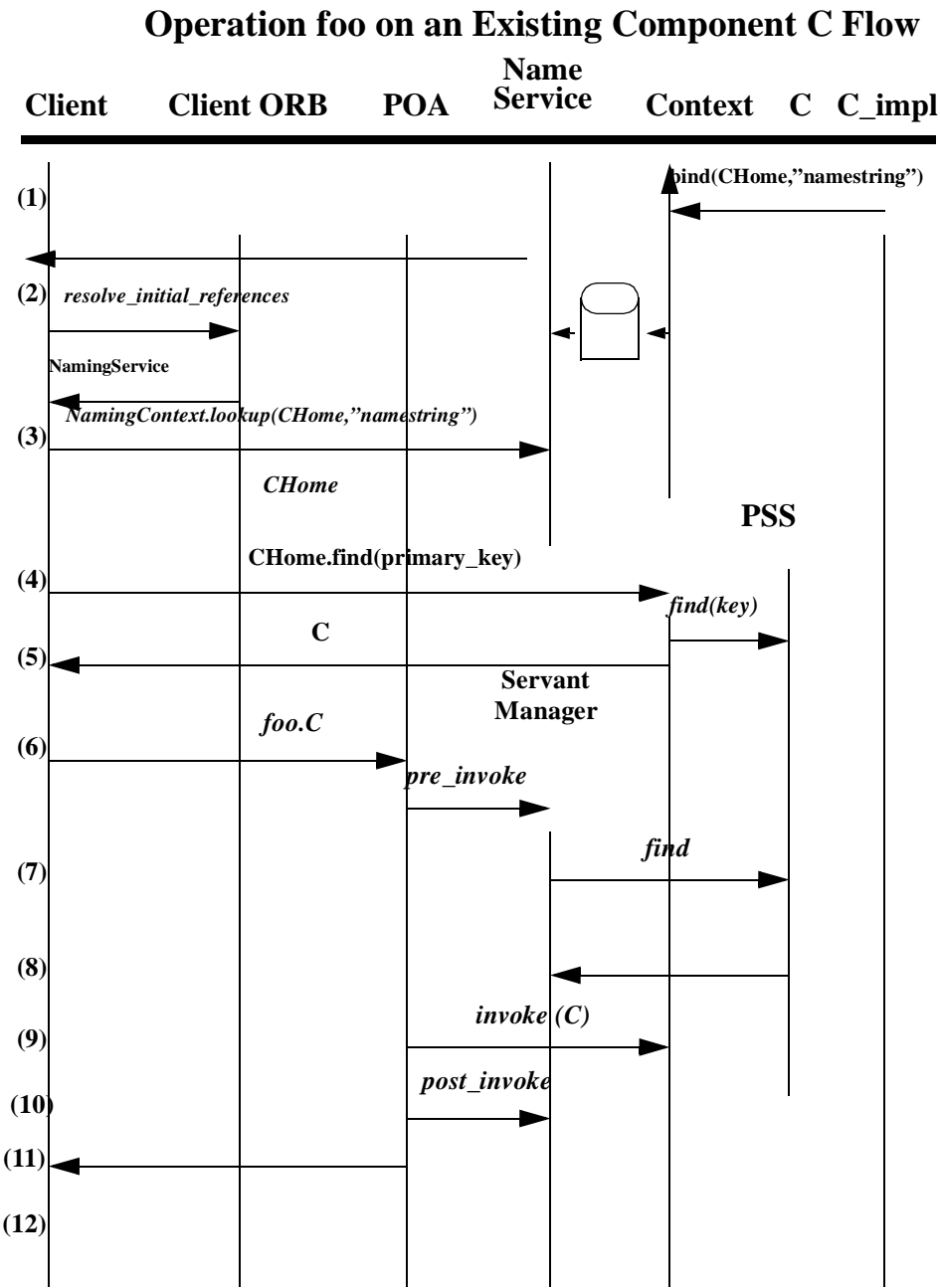


Figure 66-1 Using the Entity Container to Locate Existing Entity Components

1. Container tools binds the **entity** component home to a string (“**namestring**”) with **CosNaming**.

2. Client uses **ORB.resolve_initial_references** to get a reference to the **NamingService**. Since the **NamingContext** is a righteous CORBA object, its implementation may be located anywhere.
3. Client uses the **NamingContext.lookup** operation to find the home (**CHome**) that finds component instances of type **C**.
4. Client invokes a find operation on the home (**CHome.find**) using a primary key. Since **C** is an **entity** component, the home must talk to the persistence mechanism to locate an element in the persistent store with the same primary key.
5. A reference to **C** is returned to the client.
6. Client invokes the **foo** operation on **C** (**C.foo**). Since **C** is not active, the POA invokes the **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
7. The **ServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism find the appropriate incarnation and returns it to the **ServantLocator**.
8. The **ServantLocator** creates a new **executor** to handle the request. The associated servant is returned to the POA.
9. The POA then dispatches the request to the servant (**invoke(C)**)
10. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
11. POA returns **foo** response to client.
12. Steps [6] through [11] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor** to the pool.

66.2.5.6 *Servant Lifetime Management*

The entity container supports multiple servant lifetime policies. Support for multiple servant lifetime policies is equivalent to the process container as described in Section 66.2.4.4.

66.2.6 *The EJBSession Container*

The **EJBSession** container implements the runtime environment for a EJB Session Beans in a CORBA component container. The **EJBSession** container can be implemented using a POA with the policies enumerated in Table 66-5. These values

can be the same as a session container enabling the same POA to be used. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-5 POA Policies for a EJBSession Container

Policy Name	Required Value	Design Value
Thread		SINGLE_THREAD_MODEL ORB_CTRL_MODEL
Lifespan	TRANSIENT	
Object Id Uniqueness		N/A
Id Assignment		SYSTEM_ID USER_ID
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN
Transaction Policy	ALLOWS_SHARED	
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL_MODEL** allows the container to serialize access to Session Beans which must be single-thread. A thread policy value of **SINGLE_THREAD_MODEL** can also be used to rely on serialization in the POA, rather than the container.

Lifespan

A lifespan policy value of **TRANSIENT** is required since EJB Session Beans may have transient state and identity.

Object Id uniqueness

The object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN**.

Id assignment

An Id assignment value of **SYSTEM_ID** is sufficient for EJB Session Beans since the EJB Component Architecture does not expose object references. A value of **USER_ID** allows the container to assign unique **ObjectIds** itself. This supports a structuring of **ObjectId** values which the container can exploit within its implementation.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the servant manager.

66.2.6.1 *Creating Object References*

Object references are not exposed to the bean programmer for EJB Session Beans. Only **EJBHome** and **EJBObject** have externally visible references and they are created by the EJB container's tools, not the enterprise Bean programmer. To support a Session Bean in a CORBA EJB container, the container provider will need to do the following:

- create interface definitions for **EJBHome** and **EJBObject** and store those definitions in the interface repository.
- create entries in CORBA naming using the symbolic name defined by EJB which point to the instances of **EJBHome** and **EJBObject** to be used by this Session Bean.
- create an implementation of **EJBHome** which delegate factory operations to the enterprise Bean's create methods.
- create an implementation of **EJBObject** which delegate application operations to the enterprise Bean's application operations.

66.2.6.2 *Factories and Instances*

EJB client programmers locate factories using **JNDI**. From the EJB client's perspective, factories for Session Beans are operations implemented on **EJBHome**. The enterprise Bean developer implements the operations and the container provider stores its symbolic name in CORBA naming so it can be accessed by a client **JNDI** call and implements the **EJBHome** object which delegates to the enterprise Bean's create operation. A factory operation on **EJBHome** creates instance of the enterprise Bean which is derived from **EJBObject**. Because home operations are delegated to the enterprise Bean, **EJBHome** and **EJBObject** must be collocated.

66.2.6.3 *Invoking an Operation*

EJB clients make all operation requests on **EJBObject**. Installing a Session Bean in an **EJBSession** container requires the container to create an **EJBObject** implementation of the enterprise Bean's operations which ultimately delegates the processing of the request to the implementation. In many EJB container implementations, the **EJBObject** implementation implements the EJB container functions, including setting declarative transaction and security policies before invoking the enterprise Bean's operations.

In the CORBA environment of the exemplary design, these functions are performed by the specialized **ServantLocator** for the **EJBSession** Container before the operation request on **EJBObject** is actually dispatched by the POA. This allows the generated **EJBObject** implementation to simply delegate the operation request to the Session Bean. This is illustrated in Figure 66-1 below:

Operation foo on EJB Component J

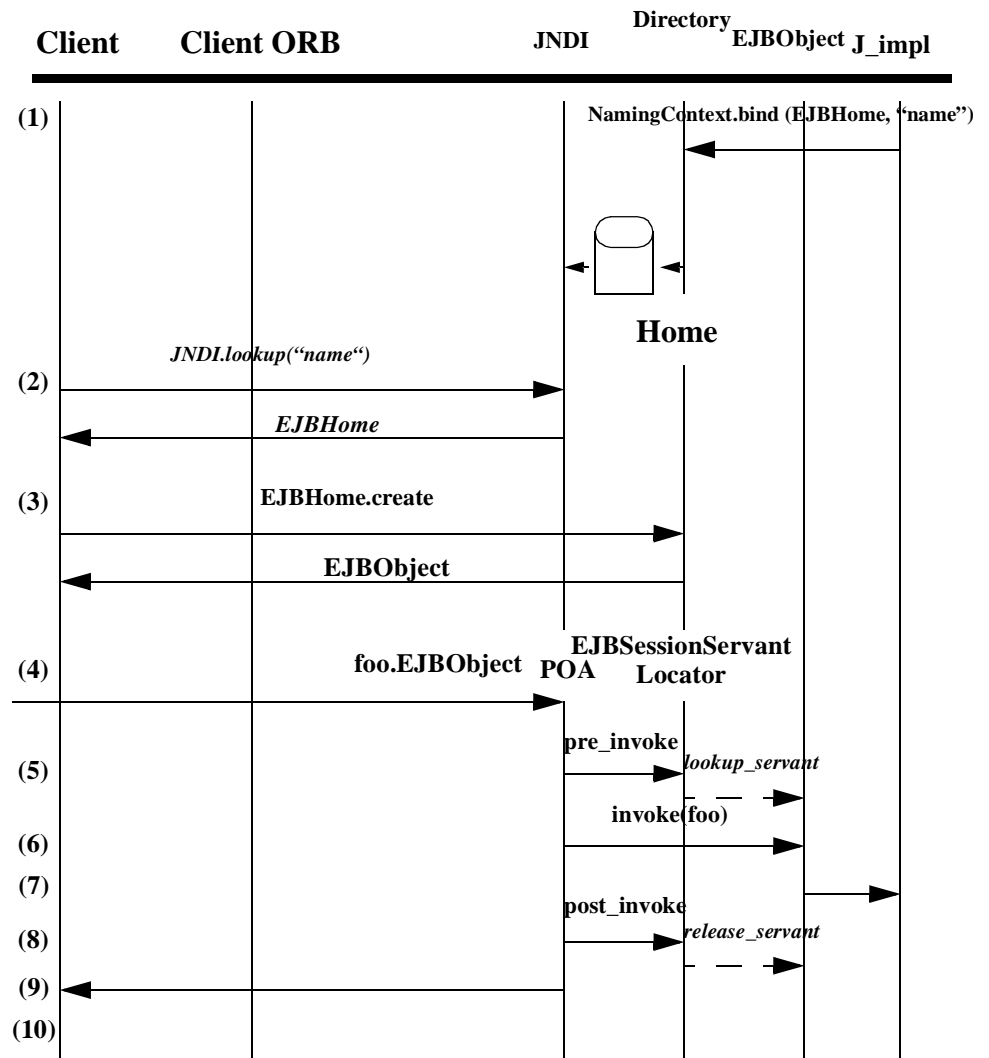


Figure 66-1 Dispatching an operation request in a CORBA EJB container

1. Container tools binds a session bean's home with CORBA naming to enable access via JNDI (**NamingContext.bind**).
2. Client uses JNDI to locate the **EJBHome** (**Jhome**) that creates component instances of type **J**.

3. Client invokes a **create** operation on the Session Bean home (**EJBHome.create**). Since **J** is a session bean, the home creates a reference and delegates the processing of the create operation to an **ejbcreate** operation of the enterprise Bean.
4. Client invokes the **foo** operation on **EJBObject** (**EJBObject.foo**).
5. The POA invokes the **ServantLocator** and requests an **executor** to process the request (**ServantLocator.pre_invoke**). The **ServantLocator** locates an appropriate **executor** or creates a new one. It returns the associated servant to the POA.
6. The POA dispatches the request to the **EJBObject** implementation (**Invoke EJBObject.foo**).
7. The **EJBObject** implementation delegates the operation to the Session Bean implementation.
8. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
9. POA then returns **foo** response back to client.
10. Steps [4] through [7] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor** to the pool.

66.2.6.4 *Servant Lifetime Management*

Enterprise JavaBeans relies on the garbage collection features of the Java language to manage bean lifetimes. This is equivalent to a servant lifetime policy of **container**.

66.2.7 *The EJBEntity Container*

The **EJBEntity** container provides the runtime environment for an EJB Entity Bean. An **EJBEntity** container can be implemented using a POA with the policies outlined in Table 66-6. These values can be made equivalent to those specified for the process and entity containers enabling the same POA to be used. Required values must be specified for all container designs. Design values correspond to the **ServantLocator** design used by the exemplary design.

Table 66-6 POA Policies for the EJBEntity Container

Policy Name	Required Value	Design Value
Thread		ORB_CTRL_MODEL SINGLE_THREAD_MODEL
Lifespan	PERSISTENT	
Object Id Uniqueness		N/A

Table 66-6 POA Policies for the EJBEntity Container

Policy Name	Required Value	Design Value
Id Assignment		USER_ID SYSTEM_ID
Implicit Activation		NO_IMPLICIT_ACTIVATION
Servant Retention		NO_RETAIN
Transaction Policy	ALLOWS_SHARED	
Request Processing		USE_SERVANT_MANAGER

Thread

A thread policy value of **ORB_CTRL__MODEL** allows the container to serialize access to Entity Beans which must be single-thread. A thread policy value of **SINGLE_THREAD_MODEL** can also be used to rely on serialization in the POA, rather than the container.

Lifespan

A lifespan policy value of **PERSISTENT** is required since Entity Beans have both persistent state and identity.

Object Id uniqueness

The object Id uniqueness policy value is not applicable when the servant retention policy is **NON_RETAIN**.

Id assignment

An Id assignment policy value of **SYSTEM_ID** is sufficient for EJB Entity Beans since the EJB Component Architecture does not expose object references. Entity Beans do support the concept of **Handle** which could be implemented as a CORBA persistent object reference. If so, a value of **USER_ID** allows the container to assign an unique **ObjectId** which can be an EJB **Handle** or some index to it.

implicit activation

The implicit activation policy must be set to **NO_IMPLICIT_ACTIVATION** when the servant retention policy is **NON_RETAIN**.

servant retention

A servant retention policy value of **NO_RETAIN** is required to use a **ServantLocator**.

transaction policy

A transaction policy value of **ALLOWS_SHARED** is required to permit the container to set transaction policy based on the component's deployment descriptor.

request processing

A request processing policy value of **USE_SERVANT_MANAGER** allows the container to be implemented in the **ServantManager**.

66.2.7.1 Creating Object References

Object references are not exposed directly to the enterprise Bean programmer for Entity Beans although they are exposed indirectly via the **Handle**. Only **EJBHome** and **EJBObject** have externally visible references and they are created by the EJB container's tools, not the enterprise Bean programmer. To support an Entity Bean in a CORBA EJB container, the container provider will need to do the following:

- create interface definitions for **EJBHome** and **EJBObject** and store those definitions in the interface repository.
- create entries in CORBA naming using the symbolic name defined by EJB which point to the instances of **EJBHome** and **EJBObject** to be used by this Entity Bean.
- create an implementation of **EJBHome** which delegate factory and finder operations to the enterprise Bean's **ejbcreate** and **ejbfind<METHOD>** operations.
- create an implementation of **EJBObject** which delegate application operations to the enterprise Bean's application operations.

66.2.7.2 Factories and New Instances

EJB client programmers locate **EJBHome** using **JNDI**. From the EJB client's perspective, factories for Entity Beans are operations implemented on **EJBHome**. The enterprise Bean developer implements the operations and the container provider implements the **EJBHome** object which delegates to the enterprise Bean's **ejbcreate** operations. The container also stores a symbolic name for **EJBHome** in CORBA naming so it can be accessed by a client **JNDI** call. A create operation on **EJBHome** creates an instance of the enterprise Bean which derives from **EJBObject**. Because home operations are delegated to the enterprise Bean, **EJBHome** and **EJBObject** must be collocated.

66.2.7.3 Invoking an Operation on a New Instance

EJB clients make all operation requests on **EJBObject**. Installing an Entity Bean in an **EJBEntity** container requires the container to create an **EJBObject** implementation of the enterprise Bean's methods which ultimately delegates the processing of the request to the bean implementation. In many EJB container implementations, the **EJBObject** implementation implements the EJB container functions, including setting declarative transaction and security policies before invoking the enterprise Bean's operations.

In the CORBA environment of the exemplary design, these functions are performed by the **ServantManager** for the **EJBEntity** Container before the operation request on **EJBObject** is actually dispatched by the POA. This allows the generated **EJBObject** implementation to simply delegate the operation request to the enterprise Bean. Figure 66-1 shows the necessary steps to make an operation request on an Entity Bean in the **EJBEntity** container:

Operation foo on a new Entity Bean K Flow

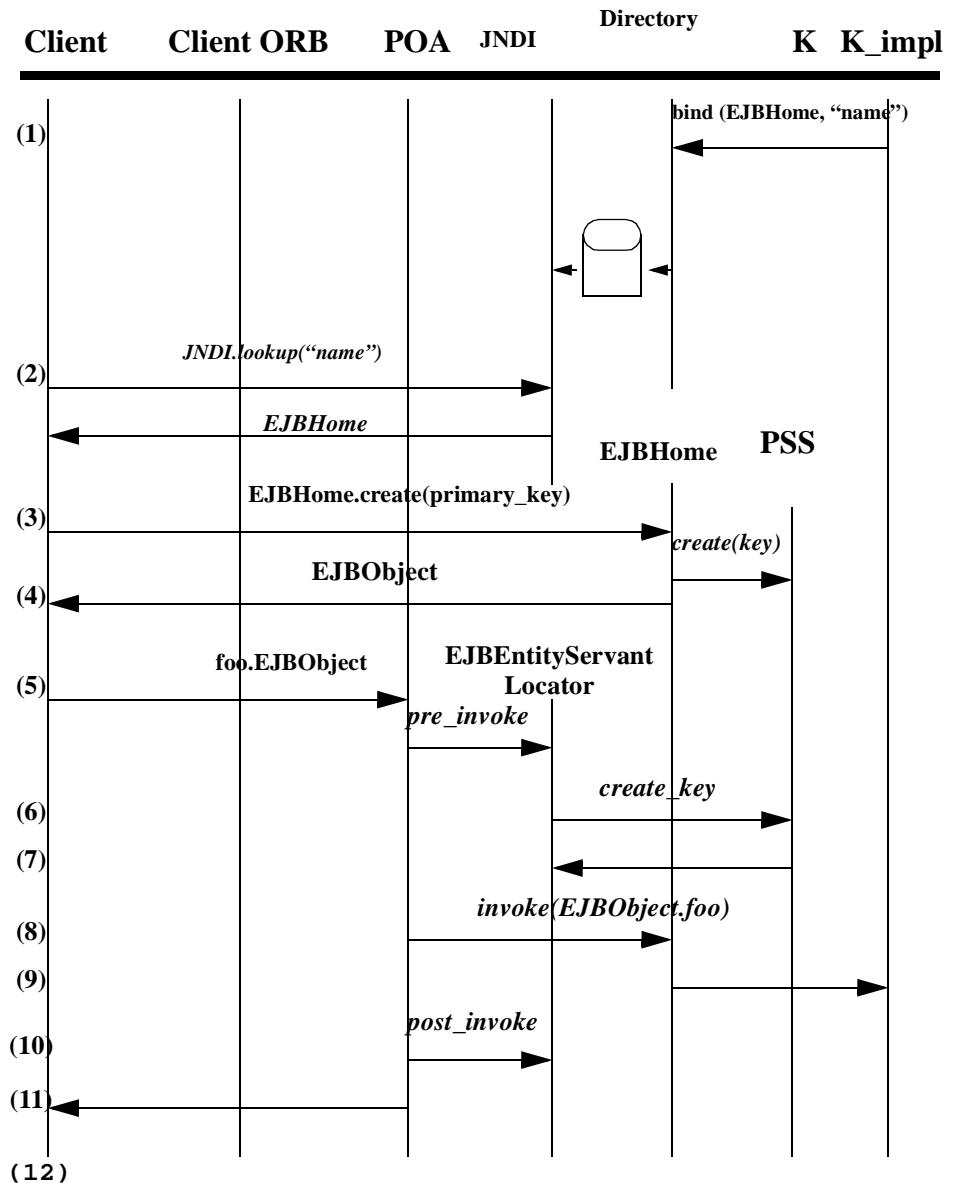


Figure 66-1 Using the EJBEntity Container to Create new Entity Beans

1. Container tools binds a entity bean's home with CORBA naming to enable access via JNDI (**NamingContext.bind**).
2. Client uses JNDI to locate the **EJBHome (Khome)** that creates component instances of type **K**.
3. Client invokes a **create** operation on the entity bean home (**EJBHome.create**) using a primary key. Since **K** is an entity bean, the home must talk to a persistence mechanism to create a new record in the persistent store using the same primary key.
4. A reference to **EJBObject** is returned to the client.
5. Client invokes the **foo** operation on **EJBObject** (**EJBObject.foo**). Since **EJBObject** is not active, the POA invokes the **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
6. The **ServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism finds the appropriate incarnation and returns it to the **ServantLocator**.
7. The **ServantLocator** creates a new **executor** to handle the request. The associated servant is returned to the POA to process the request.
8. The POA then dispatches the request to the servant (**invoke(EJBObject.foo)**).
9. The **EJBObject** implementation delegates the operation to the Entity Bean implementation.
10. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
11. The POA returns **foo** response to client.
12. Steps [5] through [11] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor**.

66.2.7.4 Finders and Existing Instances

EJB client programmers locate **EJBHome** using **JNDI**. From the EJB client's perspective, finders for entity beans are also operations implemented on **EJBHome**. The enterprise Bean developer implements the operations and the container provider implements the **EJBHome** object which delegates to the enterprise Bean's **ejbfind<METHOD>** operation. The container also stores a symbolic name for **EJBHome** in CORBA naming so it can be accessed by a client **JNDI** call. A **findByPrimaryKey** operation on **EJBHome** locates an instance of **EJBObject** using a primary key. Because home operations are delegated to the enterprise Bean, **EJBHome** and **EJBObject** must be collocated.

66.2.7.5 Invoking an Operation on an Existing Instance

Figure 66-1 shows the necessary steps to make an operation request on an existing Entity Bean in an **EJBEntity** container:

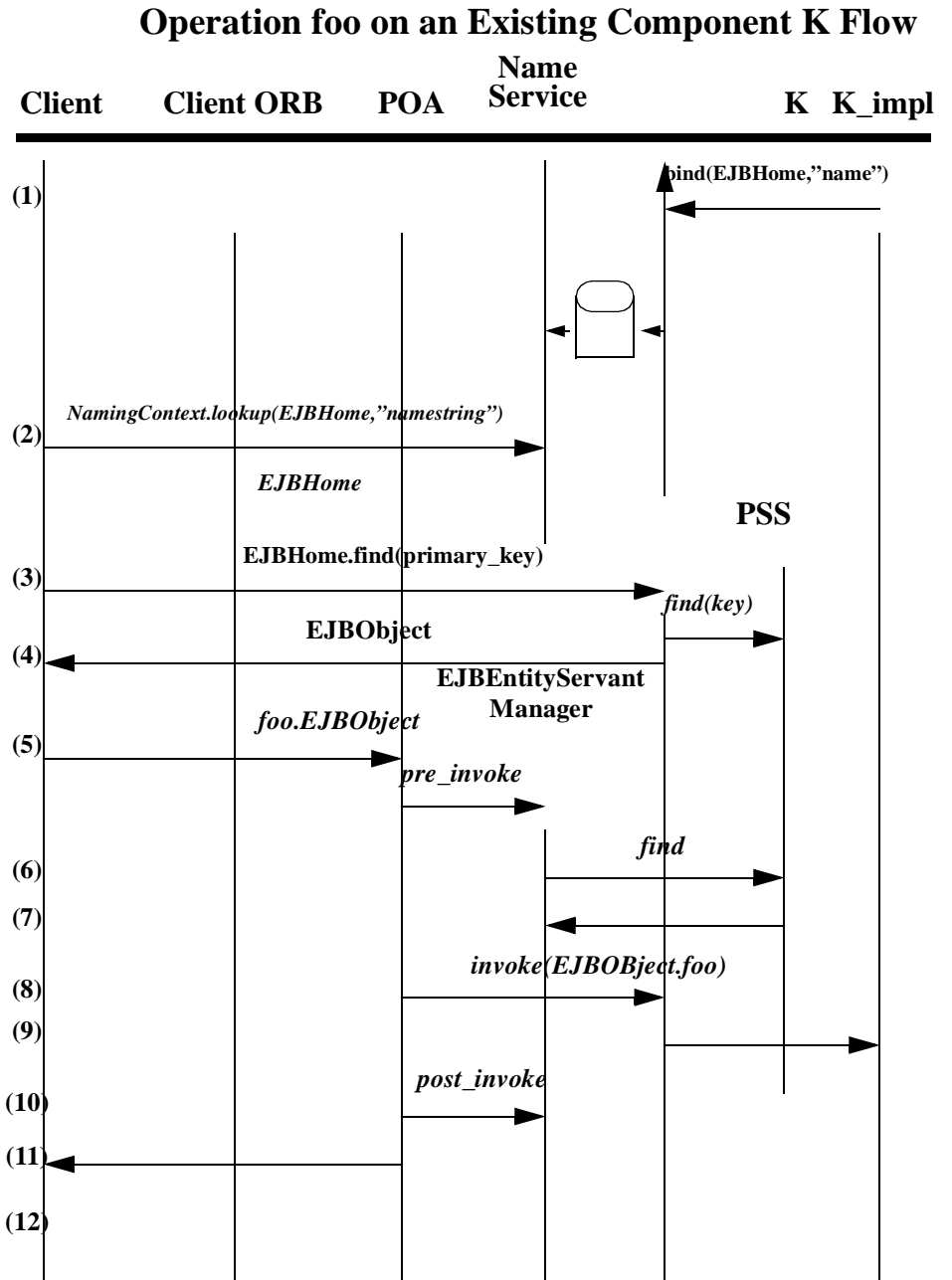


Figure 66-1 Using the EJBEntity Container to Locate Existing Entity Beans

1. Container tools binds the entity bean's home to a string ("**name**") with **CosNaming**.

2. Client uses **JNDI.lookup** operation to find the home (**EJBHome**) that finds component instances of type **K**.
3. Client invokes a find operation on the home (**EJBHome.find**) using a primary key. Since **K** is an entity bean, the home must talk to the persistence mechanism to locate an element in the persistent store with the same primary key.
4. A reference to **EJBObject** is returned to the client.
5. Client invokes the **foo** operation on **EJBObject** (**EJBObject.foo**). Since **EJBObject** is not active, the POA invokes the **pre_invoke** operation on the **ServantLocator** (**ServantLocator.pre_invoke**).
6. The **ServantLocator** talks to the persistence mechanism to find the incarnation associated with this request. The persistence mechanism find the appropriate incarnation and returns it to the **ServantLocator**.
7. The **ServantLocator** creates a new **executor** to handle the request. The associated servant is returned to the POA.
8. The POA then dispatches the request to the servant (**invoke(EJBObject.foo)**).
9. The **EJBObject** implementation delegates the operation to the Entity Bean
10. After the request completes, the POA invokes the **ServantLocator** (**ServantLocator.post_invoke**).
11. POA returns **foo** response to client.
12. Steps [5] through [11] are repeated until the operation following the expiration of the servant lifetime policy. At that point, the **ServantLocator** releases the associated **executor** to the pool.

66.2.7.6 *Servant Lifetime Management*

Enterprise JavaBeans relies on the garbage collection features of the Java language to manage bean lifetimes. This is equivalent to a servant lifetime policy of **container**. However, since entity beans are required to use transactions, the **EJBEntity** container may choose to implement a servant lifetime policy of **transaction**.

66.3 *Persistence Integration*

Component persistence is supported by the process, entity, and **EJBEntity** containers. The container architecture permits the persistence provider to be separate from the container provider since we expect that these functions will often be provided by different vendors. This section describes the various forms of persistence support available for CORBA components and the responsibilities of the container, the persistence provider, and the component developer.

Two forms of component persistence are supported by each of the containers supporting persistence:

- Container-managed persistence where the container provider interacts with the persistence provider and
- Self-managed persistence where the component developer must interact with the persistence provider.

These are described more fully in the following sections.

The process and entity containers also support a run time accessor to a set of persistence API functions, provided by the CORBA persistent state service, which enable the component to save and restore its private state. If other mechanisms are used for component persistence (e.g. SQL, ODBC, etc.), it is the responsibility of the component developer to implement the mapping directly. This is illustrated in Figure 66-1 below:

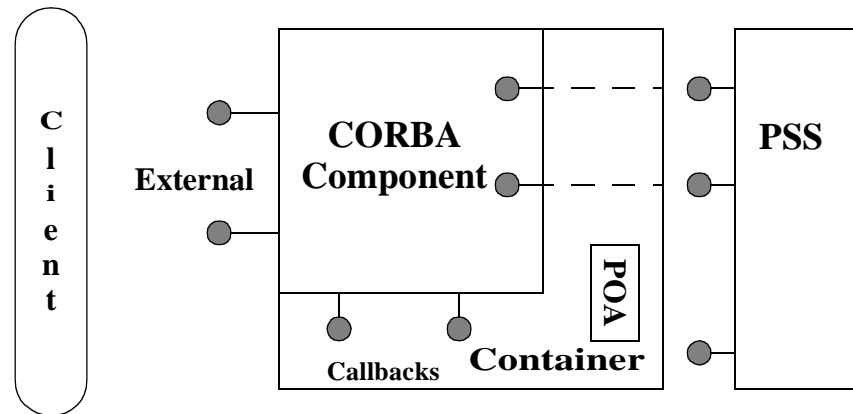


Figure 66-1 Container Persistence Architecture

The entity and **EJBEntity** containers also support access to the primary key. A **primaryKey** value is associated with the component's home for these **container categories**.

66.3.1 Container-managed Persistence

Container-managed persistence supports the declaration of abstract state associated with the component or its facets. This abstract state is declared using a state declaration language defined by the CORBA persistence state service. State which is to be container-managed can use the CORBA persistence state service or it may use some other persistence mechanism as long as that mechanism can support the persistence framework defined by the CORBA persistent state service.

When CORBA persistence is used, code can be generated to support the **ccm_load** and **ccm_store** operations on the **EntityComponent** interface or the **ejbLoad** and **ejbStore** operations of the EJB Entity container. For process and entity containers supporting extended components, this code may make use of the runtime access to the persistence provider. For basic components, access to a persistence mechanism is not specified and left to the container implementation.

This is identical to the design for EJB 1.1 which specifies the fields which participate in container-managed persistence, but leaves how those fields are made persistent to the container providers. Although common expectation is that JDBC will be used, that is not mandated.

For EJB Entity containers, it is likely that this code will utilize **JDBC** or some other Java persistence mechanism since there is not an abstract state definition language currently defined for EJB.

If CORBA persistence is not used in the process and entity containers, the component developer must implement the **ccm_load** and **ccm_store** operations as well as provide implementations for all factory and finder methods defined on the component's home.

66.3.2 Self-managed Persistence

Self-managed persistence is also supported by the same **container categories**. Self-managed persistence is assumed by process and entity containers if abstract state declarations do not exist for a particular component. With self-managed persistence, automatic code generation for saving and restoring state is not possible, so the responsibility lies completely with the component developer. Again, the developer may chose between the CORBA persistence state service and other persistence mechanisms.

For process and entity containers supporting extended components, the container provides run time access to the CORBA persistent state service which may be used. For basic components, the persistence API is the responsibility of the component implementor and is not specified. It is expected that normal database APIs such as ODBC, JDBC, or SQL will be used. Extended component developers must use the operations on **Entity2Context** to create a **ComponentId** that encapsulates the information model which describes the persistent state associated with a component. These operations are defined in Section 62.4.3.6, "ComponentId Interface," on page 62-160.

66.3.3 Interactions between the Container and the Persistence Provider

The design for CORBA components assumes the likelihood that containers and persistence solutions will be provided by different vendors. This assumption effects both the component developer and the container provider. The component developer is isolated from the persistence provider by the CORBA persistent state service which defines persistence APIs for the component developer. The container provider has several responsibilities for persistence integration. These include:

- establishing connection to the persistence mechanism,
- managing DB connections with the persistence store
- synchronizing component state with durable state.

These subjects are covered in the next sections.

66.3.3.1 *Connecting to the Persistence Mechanism*

As part of creating a container supporting persistence (process, entity, and **EJBEntity** containers), connectivity to the persistence mechanism must be established. This includes obtaining initial references the persistence provider makes available through the ORB, connecting to the persistence provider (including the exchange of security information required), and obtaining references from the persistence provider to implement an accessor to the persistence APIs provided by the CORBA persistence state service.

66.3.3.2 *Managing DB Connections*

Most persistence providers today require that a DB connection be allocated by a client before any data access operations can be invoked. Typically, this is a very expensive operation, which must be done infrequently to achieve reasonable system performance. We expect container implementations to manage a pool of such connections, which are constructed as part of the container creation process, and allocate these to component implementations as needed, typically for the duration of a transaction, although a connection may be retained longer if the container does not need it for some other component. As a result, component implementations will not have to deal with this function directly and the DB connection can be assigned to a component when its initial request to the persistence provider is made.

66.3.3.3 *Synchronization of Component State with Persistence State*

The interfaces provided by the CORBA persistent state service supports **flush** operations which can be used by the component developer to transfer state from the container domain to the persistence domain.

- For self-managed persistence, the component developer assumes this responsibility by implementing the **ccm_store** callback operation.
- For container-managed persistence, the container assumes this responsibility and invokes the **flush** operation on each persistent store involved in the current transaction.

Both approaches guarantee that the persistence provider, and not the component developer or the container, assumes the responsibility for durability of persistent state.

66.4 *Event Management Integration*

CORBA components define a simple event model which supports two forms of event communication:

- events which are published anonymously to a dedicated channel
- events which are published anonymously to a shared channel

The container is responsible for mapping those semantics onto the CORBA notification service. Although it is possible to connect event consumers and suppliers directly in some cases, the container will always deliver component events through a notification

channel to ensure a more robust event distribution mechanism and to allow consistent transaction semantics (defined with the event deployment descriptor) to be applied to both the delivery of the event to the channel and the removal of the events from the channel (i.e. a two-transaction model).

A component event is represented as a CORBA **valuetype**. This permits event emitters and publishers to be matched with their consumers by the event types they wish to exchange. The event architecture as described in Section 61.6, “Events,” on page 61-44 requires that the **valuetype** be able to be transmitted as a CORBA **any** through an event channel. This makes it possible for the container to use untyped notification channels for transmitting the actual event. The container's responsibility can be broken into three major areas and is described in the next few sections:

- setting up the channels to be used, including all the required proxies
- accepting a CORBA component event and pushing it to an event channel as a structured event
- receiving a structured event from an event channel and converting it to a CORBA component event

66.4.1 Channel setup

When a component is installed in a container, the deployment descriptor contains information about the types of events published or emitted and the types of events the component consumes. The container is responsible for initializing the CORBA notification service and establishing the event channels to be used.

- For published events, it accomplishes this with the **Event::create_channel** operation which creates a unique channel for this event type.
- For emitted events, it connects the component to a pre-configured channel which supports the **CosNotifyChannelAdmin::SupplierAdmin**.
- For consumed events, it connects the component to a pre-configured channel which supports the **CosNotifyChannelAdmin::ConsumerAdmin**.

The actual channel names are not defined in the deployment descriptors and must be made available to the container in container-specific configuration data. This allows the installation to configure shared channels to be used by other users of CORBA notification as well as component implementations. The container must create a unique channel for events which are designated as emanating from this component only. The technique by which uniqueness is ensured is not specified.

There are several possible schemes that could be made to work. Channels could be given unique names using something like a UUID to ensure uniqueness. Hierarchical names is another possibility, where all channels created by a specific container would be prefixed by the name of the container (perhaps a URL). CORBA Security could also be used to prevent events from being pushed to a channel which is dedicated to component events. Other schemes are also possible.

The CORBA notification service supports filters on both the supply side and the consume side of a channel and allows them to be configured on the channel itself, or on the proxy being used to supply or consume events. This specification allows the container provider to setup filters in any way it chooses since they too must be made available to the container at container creation time through a container-specific configuration file.

66.4.2 Transmitting an event

When a CORBA component emits or publishes an event (using the **push** operation on **<event_type>Consumer**), the operation is delegated to the container by the generated code so that the container can actually push this event to the proper channel. The following steps are required:

- channel lookup - for emitted events, this is the channel configured for general use at container start-up, for published events, this is the channel established by the container for the purpose of pushing this event type.
- Constructing the notification **EventHeader** - The **EventHeader** consists of some static information, including the two-part (**domain_name** and a **type_name**) **event_type** (not to be confused with the **<event_type>** of the CORBA **valuetype** which holds the event) and **event_name**. These fields may optionally be provided on the **Event::obtain_channel** operation. If not, they are defaulted as outlined in Table 66-7 below.
- If configuration-defined filterable data is to be associated with this event, it is placed in the portion of the structured event header defined by the CORBA notification service (**CosNotification::FilterableEventBody**). Container implementations are not required to insert filterable data.
- The **valuetype** representing the actual event data is placed into the **any** portion of the structured event.
- A **CosNotifyComm::push_structured_event** is issued to CORBA notification.

Table 66-7 Structured Event Header

Event Header field	Default Value
domain_name	CCM or blank
type_name	Repository Id of <event_type>
event_name	blank
filterable_data	blank

66.4.3 Receiving an event

In order to receive an event, the container must connect its proxy to the event channel the event is to be received on and implement the **CosNotifyComm::structured_push_consumer** interface. The container connects to the channel as a result of an **Event::listen** operation. The container performs a

CosNotifyChannelAdmin::connect_structured_push_consumer operation on behalf of the component. The **listen** operation receives all events from the channel, subject to filter constraints.

When the container's **structured_push_consumer** interface is invoked, it performs the following processing:

- It extract the event data from the **any** portion of the structured event and converts it to a CORBA **valuetype** which represents the event.
- It removes the **domain_name**, **type_name**, and **event_name** from the **EventHeader**.
- It extracts the **<event_type>** from the component event in the **any** portion of the event data structure.
- It invokes **<event_type>Consumer::push** passing in the **valuetype <event_type>**.

Packaging and Deployment

69

This chapter describes the CORBA component **packaging and deployment model**.

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

69.0.0.1 Contents

This chapter contains the following sections.

Section Title	Page
“Introduction”	69-258
“Component Packaging”	69-258
“Software Package Descriptor”	69-259
“CORBA Component Descriptor”	69-273
“Component Assembly Packaging”	69-298
“Component Assembly File”	69-298
“Component Assembly Descriptor”	69-298
“Property File Descriptor”	69-321
“Component Deployment”	69-327

69.1 Introduction

Component implementations may be packaged and deployed.

A CORBA Component package maintains one or more implementations of a component. It may be installed on a computer or grouped together with other components to form an *assembly*. A component assembly is a group of interconnected components represented by an assembly package.

A package, in general, consists of one or more descriptors and a set of files. The descriptors describes the characteristics of the package and points to its various files. The files that make up a package, including the descriptor, may be grouped together in an archive file or stored separately. When stored separately, the descriptor contains pointers to the location of each file.

The component package is a specialization of a general software package. The software packaging scheme, described here, could be used to package arbitrary software entities. In fact it was initially inspired by the Open Software Description (OSD) note to the W3C. OSD is an XML vocabulary for describing software packages and their dependencies. We have extended OSD slightly, without loss of generality, to support component packaging.

A component package may be deployed alone, as is, or it may be included in a component assembly package and deployed as part of the assembly along with the other components of the assembly.

A component assembly is a set of interrelated components and component homes represented by an assembly package. A component assembly package consists of a set of component packages and an assembly descriptor. The assembly descriptor specifies the components that make up the assembly, partitioning constraints, and connections. Connections are between interface ports, represented by *provides* and *uses* features and between event ports, represented by *emits*, *produces*, and *consumes* features.

Component and assembly packages are provided as input to a deployment tool.

A deployment tool deploys individual components and assemblies of components to an installation site, usually a set of hosts on a network. The user of the deployment tool guides in determining where each component should be installed. Components within an assembly may be installed on a single machine or scattered across a network.

Based on an assembly descriptor and user input, the deployment tool installs and activates component homes and instances; it configures component properties and connects components together via interface and event ports, as indicated in the assembly descriptor.

69.2 Component Packaging

A software package is represented by a descriptor and a set of files. The descriptor and associated files are grouped together in a ZIP archive file. The software package could be used to describe arbitrary software packages.

In relation to CORBA Components, software packages are used to package a CORBA Component implementation.

69.3 Software Package Descriptor

The contents of a software package is described by a software package descriptor. The descriptor consists of general information about the software followed by one or more sections describing implementations of that software. An XML vocabulary is used to describe component software packages. The descriptor file has a “.csd” extension. CSD stands for CORBA Software Descriptor. When used in an archive, the CSD file for the archive is placed in a top level directory called “meta-inf”.

The structure and intent of the descriptor can be better understood by looking at an example.

69.3.1 A softpkg Descriptor Example

```

<softpkg name="Bank" version="1,0,1,0">
  <pkgtype>CORBA Component</pkgtype>
  <title>Bank</title>
  <author>
    <company>Acme Component Corp.</company>
    <webpage href="http://www.acmecomponent.com/">
  </author>
  <description>Yet another bank example</description>
  <license href="http://www.acmecomponent.com/license.html" />
  <idl id="IDL:M1/Bank:1.0" ><link href="ftp://x/y/Bank.idl"/></idl>

  <propertyfile><fileinarchive name="bankprops.cpf"/></propertyfile>

  <implementation id="DCE:700dc518-0110-11ce-ac8f-0800090b5d3e">
    <os name="WinNT" version="4,0,0,0" />
    <os name="Win95" />
    <processor name="x86" />
    <compiler name="MyFavoriteCompiler" />
    <programminglanguage name="C++" />

    <dependency type="ORB"><name>ExORB</name></dependency>

    <descriptor type="CORBA Component">
      <fileinarchive>processcontainer.ccd</fileinarchive>
    </descriptor>

    <code type="DLL">
      <fileinarchive name="bank.dll"/>
      <entrypoint>createBankHome</entrypoint>
    </code>

    <dependency type="DLL">
      <localfile name="rwthr.dll"/>
    </dependency>

  </implementation>

  <implementation id="DCE:297f3e18-0110-11ce-ac8f-08074982ad3e"
    variation="RemoteHome">
    <os name="Solaris" version="5,5,0,0" />
    <processor name="sparc" />
    <!-- ... -->
  </implementation>

  <implementation> <!-- another implementation --> </implementation>
</softpkg>

```

69.3.2 The Software Package Descriptor XML Elements

This section describes the XML elements that make up a software package descriptor. The section is organized starting with the root element of the package descriptor document, **softpkg**, followed by all subordinate elements, in alphabetical order. The complete **softpkg** DTD may be found in Section 695.1, “softpkg.dtd,” on page 695-335.

Note – An effective strategy for studying an XML DTD is to recursively navigate from the root element, which in this case is **softpkg**, to each child element.

69.3.2.1 The softpkg Root Element

The **softpkg** element is the root element of the document. As well, it is a child element of **dependency**. It contains a set of general child elements that describe the software package. This is followed by one or more implementation specifications.

A **softpkg** archive may contain multiple implementations of a component. This allows the component implementor to provide specialized implementations for different operating systems, compilers, or ORBs, or to provide different programming language implementations of the component. Each implementation is represented in the **softpkg** descriptor as a distinct implementation element.

```
<!ELEMENT softpkg
( title
| pkgtype
| author
| description
| license
| idl
| propertyfile
| dependency
| descriptor
| implementation
| extension
)* >

<!ATTLIST softpkg
name ID #REQUIRED
version CDATA #OPTIONAL >
```

The attributes are as follows:

- name**
Uniquely identifies the package within the package.
- version**
Specifies the version of the component. The format of the version string is numerical major and minor version numbers separated by commas (e.g., “1,0,0,0”).

69.3.2.2 *The author Element*

The **author** element is used to identify the author of the **softpkg**. It may contain **name**, **company**, and **webpage** child elements.

```
<!ELEMENT author
  ( name
    | company
    | webpage
  )* >
```

69.3.2.3 *The code Element*

The **code** element points to a file in the archive which implements the component. This could be, for example, a DLL, a .so, or a .class file. The **fileinarchive** child element is used to indicate the code file within the archive. **codebase** and **link** are used to point to code files outside of any archive. The optional **entrypoint** child element is used to specify an entry point to the code. The optional **usage** element is used to describe how to use, i.e., invoke, the code.

```
<!ELEMENT code
  ( ( codebase
    | fileinarchive
    | link
    )
    , entrypoint?
    , usage?
  ) >
<!ATTLIST code
  type CDATA #IMPLIED >
```

The **type** attribute specifies the type of code. The types “**DLL**”, “**Executable**”, and “**Java Class**” shall be recognized as valid types.

69.3.2.4 *The codebase Element*

The **codebase** element is used to specify a resource. If the resource isn’t available in the local environment, then a link specifies where it may be obtained. **codebase** has an EMPTY content model.

```
<!ELEMENT codebase EMPTY >
<!ATTLIST codebase
  filename CDATA #IMPLIED
  %simple-link-attributes; >
```

codebase has two attributes: **name** - the name of the resource, and **href**--as defined in **simple-link-attributes**--the link.

69.3.2.5 *The company Element*

The **company** element, an optional child element of **author**, specifies the company that created the **softpkg**. It contains string data.

```
<!ELEMENT company ( #PCDATA ) >
```

69.3.2.6 *The compiler Element*

The optional **compiler** element specifies the compiler used to create an implementation. **compiler** has an empty content model.

```
<!ELEMENT compiler EMPTY >
<!ATTLIST compiler
    name CDATA #REQUIRED
    version CDATA #IMPLIED >
```

The required attribute **name**, specifies the name of the compiler and the optional **version**, the version of the compiler. The version is specified in a “w,x,y,z” format.

69.3.2.7 *The dependency Element*

The **dependency** element is used to specify environmental or other dependencies. The type of dependency is specified by the **type** attribute. The **dependency** element is a child element of both the **softpkg** element and **implementation** elements. When used as a child of **softpkg**, it specifies general dependencies applicable to all implementations. When used as a child of **implementation**, it specifies implementation specific dependencies.

```
<!ELEMENT dependency
    ( softpkgref
    | codebase
    | fileinarchive
    | localfile
    | name
    ) >
<!ATTLIST dependency
    type CDATA #IMPLIED
    action (assert | install)"assert">
```

The **type** attribute specifies the type of the resource required. This may be set to, for example, “DLL”, “.so”, or “.class”.

When **action** is set to **assert**, the installation process must verify that the dependency exists in the environment. If **action** is set to **install**, the installation process must install the dependency if it does not already exist.

69.3.2.9 The descriptor Element

69.3.2.10 The entryptpoint Element

<!ELEMENT entrypt (#PCDATA) >

69.3.2.11 *The extension Element*

The **extension** element is used to add experimental or vendor specific elements to the softpkg DTD. The content model of the extension element is **PCDATA**, meaning that it can have character data or markup.

An effort has been made to make the **extension** element an optional child element of all non-trivial elements. Processors may ignore **extension** elements that they do not recognize.

<!ELEMENT extension (#PCDATA) >

<!ATTLIST extension

| | | |
|------------------|--------------|------------------|
| class | CDATA | #REQUIRED |
| origin | CDATA | #REQUIRED |
| id | ID | #IMPLIED |
| extra | CDATA | #IMPLIED |
| html-form | CDATA | #IMPLIED |

The attributes of the **extension** element are as follows:

class

Used to distinguish this extension element usage. A processing application identifies extension elements that it understands by examining an extension element's **class** and **origin** attributes.

origin

An **origin** attribute is required to identify the party responsible for the extension; for example, an ORB vendor.

id

An optional ID attribute which must be unique in the file.

extra

An *extra* attribute that may be used however the originator wishes.

html-form

The **html-form** element is used for formatting. The content will be formatted per the html element type indicated, e.g., "".

69.3.2.12 *The fileinarchive Element*

The **fileinarchive** element is used to specify a file in the same archive as the descriptor. The optional **link** element may be used to point to an external archive, in which case the file will be looked for in that file.

```
<!ELEMENT fileinarchive
    ( link? ) >
<!ATTLIST fileinarchive
    name CDATA #REQUIRED >
```

The **name** attribute specifies the name or path of the element in the archive.

69.3.2.13 *The humanlanguage Element*

The **humanlanguage** element specifies a spoken language. **humanlanguage** has an EMPTY content model.

```
<!ELEMENT humanlanguage EMPTY >
<!ATTLIST humanlanguage
    name CDATA #REQUIRED >
```

The human language name is specified in the **name** attribute.

69.3.2.14 *The idl Element*

The **idl** element points to file or repository containing an idl definition.

```
<!ELEMENT idl
    ( link
    | fileinarchive
    | repository
    ) >
```

69.3.2.15 *The implementation Element*

The **implementation** element contains descriptive information about a particular implementation of the software represented by the **softpkg** descriptor. An implementation is described by platform dependencies, descriptors, dependencies, code filename, entry points and other characteristics.

```

<!ELEMENT implementation
    ( description
    | code
    | compiler
    | dependency
    | descriptor
    | extension
    | programminglanguage
    | humanlanguage
    | os
    | propertyfile
    | processor
    | runtime
    )* >

<!ATTLIST implementation
    id ID #IMPLIED >

```

The **id** attribute is a DCE UUID which uniquely identifies the implementation.

The **variation** attribute is used to indicate a variation from a normal implementation. The interpretation of the variation attribute depends on user of the softpkg.

Note – The only valid variation string defined by the CORBA Component model is “**ProxyHome**”. The ProxyHome variation indicates that the component implementation contains a proxy home only, not a full component implementation.

69.3.2.16 *The implref Element*

The **implref** element is used to refer to an implementation within a softpkg.

```

<!ELEMENT implref EMPTY >
<!ATTLIST implref
    idref CDATA #REQUIRED >

```

The **idref** attribute refers to a unique **implementation** element **id** in the softpkg descriptor.

69.3.2.17 *The license Element*

The **license** child element of **softpkg** is used to point to the text of a usage license. The license is pointed to by an **href** attribute. The **license** element may have arbitrary string content.

```

<!ELEMENT license ( #PCDATA ) >
<!ATTLIST license
    %simple-link-attributes; >

```

69.3.2.18 *The link Element*

The **link** element is used to specify a generic link. The **href** attribute indicates the link. The element can have string content.

```

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
    %simple-link-attributes; >

```

69.3.2.19 *The localfile Element*

The **localfile** element is used to specify a file that is expected to be found in the local environment.

```

<!ELEMENT localfile EMPTY >
<!ATTLIST localfile
    name CDATA #REQUIRED >

```

The name of the file is specified in the **name** attribute.

69.3.2.20 *The name Element*

The **name** element, as an optional child element of **author**, specifies the name of the author. It has string content.

```

<!ELEMENT name ( #PCDATA ) >

```

69.3.2.21 *The os Element*

The **os** element is used to specify a particular operating system that the implementation will work with. This can be specified multiple times if the implementation will work on more than one **os**.

```

<!ELEMENT os EMPTY >
<!ATTLIST os
    name CDATA #REQUIRED
    version CDATA #IMPLIED>

```

The **name** attribute specifies the name of the operating system.

The **version** attribute specifies the version of the **os** in “w,x,y,z” format.

Legal values include:

- AIX

- BSDi
- VMS
- DigitalUnix
- DOS
- HPBLS
- HPUX
- IRIX
- Linux
- MacOS
- OS/2
- AS/400
- MVS
- SCO CMW
- SCO ODT
- Solaris
- SunOS
- UnixWare
- VxWorks
- Win95
- WinNT

69.3.2.22 *The pkgtype Element*

The **pkgtype** element is used to identify the type of software that the **softpkg** represents. This specification reserves package types “**CORBA Component**” and “**CORBA Interface Impl**” for the packaging of CORBA component and interface implementations.

```
<!ELEMENT pkgtype ( #PCDATA ) >
<!ATTLIST pkgtype
  version CDATA #IMPLIED >
```

The optional **version** attribute specifies a version of the package type.

69.3.2.23 *The processor Element*

The **processor** element indicates the type of processor that the implementation must run on, if there is any such constraint.

```

<!ELEMENT processor EMPTY >
<!ATTLIST processor
    name CDATA #REQUIRED >

```

The name of the processor is indicated in the **name** attribute.

Legal values include:

- x86
- mips
- alpha
- ppc
- sparc
- 680x0
- vax
- AS/400
- S/390

69.3.2.24 *The programminglanguageElement*

The **programminglanguage** element specifies the type of the component implementation. **programminglanguage** has an empty content model. **programminglanguage** is a child element of **implementation**.

```

<!ELEMENT programminglanguage EMPTY>
<!ATTLIST programminglanguage
    name CDATA #REQUIRED
    version CDATA #IMPLIED >

```

The required programminglanguage **name** and optional **version** attributes specify the programming language used to implement the component.

69.3.2.25 *The propertyfile Element*

The **propertyfile** element is used to refer to a property file associated with the **softpkg** or **implementation**.

A property file of a particular type, defined at the top level of the descriptor, may be overridden by implementation specific property files of that type, defined in an **implementation** element.

```

<!ELEMENT propertyfile
    ( fileinarchive
      | link ) >
<!ATTLIST propertyfile
    type CDATA #IMPLIED >

```

The **type** attribute, distinguishes a property file from other types of property files. If there is only one type of property file, or if the type of property file is implicit given a context, then the **type** is not required.

69.3.2.26 *The runtime Element*

The **runtime** element specifies a runtime required by a component implementation. An example of a runtime is a Java VM.

```

<!ELEMENT runtime EMPTY >
<!ATTLIST runtime
    name CDATA #REQUIRED
    version CDATA #IMPLIED>

```

The name and version of the runtime are specified in the **name** and **version** attributes. The version is specified in “w,x,y,z” format.

69.3.2.27 *The simple-link-attributes Entity*

The **simple-link-attributes** entity is used to specify link attributes. The default link form is a simple link.

```

<!ENTITY % simple-link-attributes "
    xml:link CDATA #FIXED 'SIMPLE'
    href CDATA #REQUIRED
">

```

The user of an element that uses these link attributes will likely only need to be concerned with the **href** attribute. However the user may specify other attributes if desired.

Note – In the context of CORBA Components, the **href** attribute may be used to specify INS format names.

To demonstrate the usage of an element that employs the **simple-link-attributes** entity, consider the following element definition:

```

<!ELEMENT exampleelement EMPTY >
<!ATTLIST exampleelement
    %simple-link-attributes; >

```

This could be used as follows:

```
<exampleelement href="http://www.abc.com/xyz" />
```

Issue – The W3C XLL work is still in progress at the time of this writing. This entity definition will be modified if necessary when the W3C work completes.

69.3.2.28 *The softpkg Element*

This is the root element of the descriptor. See section 69.3.2.1 on page 261.

69.3.2.29 *The softpkgref Element*

The **softpkgref** element refers to an external softpkg. The file is referenced by a **fileinarchive** element or a **link**. An optional **implref** element refers to a particular implementation within the softpkg descriptor.

```
<!ELEMENT softpkgref
    ( ( fileinarchive
      | link
      )
      , implref?
    ) >
```

69.3.2.30 *The title Element*

The **title** element is used to specify the friendly, or tool name of the **softpkg**. The title element contains string data.

```
<!ELEMENT title ( #PCDATA ) >
```

69.3.2.31 *The usage Element*

The **usage** element contains a string usage description.

```
<!ELEMENT usage ( #PCDATA ) >
```

69.3.2.32 *The webpage Element*

The **webpage** element, an optional child element of **author**, specifies a web page associated with the author.

```
<!ELEMENT webpage ( #PCDATA ) >
<!ATTLIST webpage
    %simple-link-attributes; >
```


69.4 CORBA Component Descriptor

The CORBA Component descriptor describes a component. It is referred to by a **<descriptor type="CORBAComponent">** element in a softpkg descriptor describing a CORBA component. The CORBA Component descriptor specifies component characteristics, used at design and deployment time. A component descriptor file has a recommended ".ccd" extension, standing for CORBA Component Descriptor.

The component descriptor is generated by a CIDL compiler. This is convenient as the CIDL compiler has much of the necessary information at hand. However, the compiler doesn't have all of the information required. The user, likely with the help of a packaging tool, will have to modify the generated descriptor. This could be done manually, but it is more likely to be done with the help of a packaging tool.

The component descriptor is described using an XML vocabulary. The complete XML DTD for the descriptor is in Appendix 695.2 on page 339. This chapter will discuss each element of the descriptor in detail.

69.4.1 Component Feature Description

The component descriptor provides information that a design tool may use to display information about a component. This includes information about the interfaces that the component supports and its ports.

Note – For the purpose of component packaging and deployment we will use the term *ports* to collectively describe the interfaces that a component uses and provides and the events that it emits, publishes, and consumes. In addition, provides and uses ports will be called *interface ports*, and emits, publishes, and consumes ports will be termed *event ports*.

The component descriptor describes the structure of a component with respect to supported interfaces, inherited components, and uses and provides ports. The component is described by a **componentfeatures** element, which describes inherited components, supported interfaces, used and provided interfaces, and emitted, published and consumed events. If the component inherits from other components then the features of that component are described in a separate **componentfeatures** element and referenced by the **inheritscomponent**. The primary **componentfeatures** element of the descriptor is indicated by the **repositoryid** element of the component descriptor.

Each interface supported or provided by a component is described by an **interface** element. Interface elements are referenced by the repository id of the interface. An **interface** has a name and a repository id, and may inherit from other interfaces. The inheritance relationship is represented by the **inheritsinterface** element.

This information allows a tool to display the features of a component and to connect components together based on those features. For example, a component which *uses* interface *X* could be *connected* to another component that *provides* interface *X*, based on information in each component's descriptor.

69.4.2 Deployment Information

At deployment time, the component descriptor is used to determine the type of container in which the component needs to be installed and to provide information about the component to the container.

The **componentkind** element tells the creator of the container what kind of container to create. A **componentkind** can be either **session**, **service**, **process**, or **entity**.

The **transaction** element indicates the transactional characteristic of the component.

The **eventpolicy** is used to indicate the quality of service of event ports

The **threading** element indicates how the container should dispatch operations on the component instance. If **threading** is set to *multithread* then the component is ready to accept multiple threads of control within a single instance. The component takes responsibility for protecting its internal state. If **threading** is set to *serialize* then the container will serialize all calls to a single instance. Note that although the component will not need to protect instance state, the container may employ other threads to invoke other instances of the component type, thus the component must protect any static or class data.

The **configurationcomplete** element tells the deployment agent whether the component expects for **configuration_complete** to be called after its properties have been set and its ports configured to their initial state (e.g., as described by a component assembly descriptor).

The **segments** element provides the container with information necessary to map segment tags to segment names, segment tags to facet tags, and segment tags to abstract storage home types. The **facettag** element references a **provides** interface element described elsewhere in the descriptor. The **provides** element maps facet tags to provided interface names. A container uses the information provided by these elements to construct data structures mapping segment tags to segment names, facet tags to facet names, and segment tags to facet tags. Note that a segment tag can map to more than 1 facet tag.

69.4.3 CIDL Compiler Responsibilities

A CIDL compiler is responsible for generating an initial component descriptor. This initial descriptor is vendor specific and may be manipulated directly by the user or using vendor supplied tools.

69.4.4 CORBA Component Descriptor Example

```

<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "corbacomponent.dtd">

<corbacomponent>
  <corbaversion> 3.0 </corbaversion>
  <componentrepid repid="IDL:BookStore:1.0" />
  <homerepid repid="IDL:BookStoreHome:1.0" />
  <componentkind>
    <entity>
      <servant lifetime="process" />
    </entity>
  </componentkind>
  <security rightsfamily="corba" />
  <threading policy="multithread" />
  <configurationcomplete set="true" />

  <segment name="bookseg" segmenttag="1">
    <segmentmember facettag="1" />
    <segmentmember facettag="2" />
    <containermanagedpersistence>
      <storagehome id="PSDL:BookHome:1.0" />
      <pssimplementation id="ACME-PSS" />
      <catalog type="PSDL:BookCatalog:1.0" />
      <accessmode mode="READ_ONLY" />
      <psstransaction policy="TRANSACTIONAL" >
        <psstransactionisolationlevel level="SERIALIZABLE" />
      </psstransaction>
    </containermanagedpersistence>
    <params>
      <param name="x" value="1" />
    </params>
  </segment>

  <homefeatures
    name="BookStoreHome"
    repid="IDL:BookStoreHome:1.0">
    <operationpolicies>
      <operation name="*">
        <transaction use="never" />
      </operation>
    </operationpolicies>
  </homefeatures>

  <componentfeatures name="BookStore" repid="IDL:BookStore:1.0">
    <inheritscomponent repid="IDL:Acme/Store:1.0" />
    <ports>
      <provides>
        providesname="book_search"
        repid="IDL:BookSearch:1.0"
        facettag="1">

```

```

    <operationpolicies>
      <operation name="getByAuthor">
        <requiredrights>
          <right name="get"/>
        </requiredrights>
      </operation>
      <operation name="getByTitle">
        <requiredrights>
          <right name="get"/>
        </requiredrights>
      </operation>
      <operation name="getByISBN">
        <requiredrights>
          <right name="get"/>
        </requiredrights>
      </operation>
    </operationpolicies>
  </provides>
  <provides
    providesname="shopping_cart"
    repid="IDL:CartFactory:1.0"
    facettag="2" />
  <uses
    usesname="ups_rates"
    repid="IDL:ShippingRates:1.0" />
  <uses
    usesname="fedex_rates"
    repid="IDL:ShippingRates:1.0" />
  <emits
    emitsname="low_stock"
    eventtype="StockRecord">
      <eventpolicy policy="normal" />
    </emits>
  <publishes
    publishesname="offer_alert"
    eventtype="SpecialOffer">
      <eventpolicy policy="normal" />
    </publishes>
  </ports>
</componentfeatures>

<componentfeatures name="Store" repid="IDL:Acme/Store">
  <supportsinterface repid="IDL:Acme/GeneralStore">
    <operationpolicies>
      <operation name="*">
        <transaction use="required" />
      </operation>
    </operationpolicies>
  </supportsinterface>
  <ports>
    <provides

```

```

        providesname="admin"
        repid="IDL:Acme/StoreAdmin:1.0"
        facettag="3" />
    </ports>
</componentfeatures>

<interface name="BookSearch" repid="IDL:BookSearch:1.0">
    <inheritsinterface repid="IDL:SearchEngine:1.0" />
</interface>
<interface name="SearchEngine" repid="IDL:SearchEngine:1.0"/>
<interface name="CartFactory" repid="IDL:CartFactory:1.0"/>
<interface name="ShippingRates" repid="IDL:ShippingRates:1.0"/>
<interface name="StoreAdmin" repid="IDL:Acme/StoreAdmin:1.0">
    <operationpolicies>
        <operation name="*">
            <transaction use="required" />
            <requiredrights>
                <right name="manage"/>
                <right name="set"/>
            </requiredrights>
        </operation>
    </operationpolicies>
</interface>
<interface name="GeneralStore" repid="IDL:Acme/GeneralStore:1.0"/>

</corbacomponent>

```

69.4.5 The CORBA Component Descriptor XML Elements

This section describes the XML elements that make up a component descriptor. The section is organized starting with the root element of the component descriptor document, **corbacomponent**, followed by all subordinate elements, in alphabetical order. The complete CORBA component descriptor DTD may be found in Section 695.2, “corbacomponent.dtd,” on page 695-339.

69.4.5.1 The corbacomponent Root Element

The **corbacomponent** element is the root element of the CORBA component descriptor.

```
<!ELEMENT corbacomponent
    ( corbaversion
      , componentrepid
      , homerepid
      , componentkind
      , interop?
      , transaction?
      , security?
      , threading
      , configurationcomplete
      , extendedpoapolicy*
      , repository?
      , segment*
      , componentproperties?
      , homeproperties?
      , homefeatures+
      , componentfeatures+
      , interface*
      , extension*
    ) >
```

These elements must be provided in the order presented.

- **corbaversion** tells which version of CORBA the component is assuming.
- **componentrepid** is the interface repository id of the component. It also refers to a **componentfeatures** element later in the descriptor.
- **homerepid** is the interface repository id of the home. It also refers to a **homefeatures** element later in the descriptor.
- **componentkind** describes properties of the component which will determine what kind of container the component must reside in.
- **interop** specifies interoperation information, e.g., with EJB.
- **transaction** determines transaction policies for the entire component. This policy is optional and may be overridden on individual facets or supported interfaces.
- **security** specifies CORBA security rights family for the component.
- **threadingpolicy** determines whether calls to the component will be serialized or not.
- **configurationcomplete** is set if the component expects for **configuration_complete** to be called on the component after all of its properties have been set and its ports have been connected.
- **extendedpoapolicy** is used to set a POA policy for the component beyond the base POA policies. For example, firewall policies.

- **repository** provides a reference to a repository, such as the interface repository.
- **segment** describes a segment including its name, tag, member facets, and storage home type.
- **homefeatures** describes the structure of the component's homes.
- **componentproperties** specifies the default component properties file.
- **homeproperties** specifies the default home properties file.
- **componentfeatures** describes inherited components, supported interfaces, uses and provides ports, and emits, publish, and consumes ports of the component. If the primary component inherits from other components, those components are described in separate **componentfeature** elements.
- **interface** describes the simple name and repository id of an interface and points to inherited interfaces. Between the **componentfeatures** and **interface** elements, one can navigate all of the interfaces that a component uses, provides, supports, and inherits.
- **extension** may be used by a user or vendor to provide proprietary information in the component descriptor.

These are the top-level elements of the document. These descriptor elements are described in terms of attributes and other elements. The remainder of this section will describe the top-level and child elements in detail.

Elements are presented in alphabetical order so that they will be easy to locate.

See Section 695.2, "corbacomponent.dtd," on page 695-339 for the full text of the component descriptor DTD.

69.4.5.2 *The accessmode Element*

Child element of **containermanagedpersistence**.

The **accessmode** element identifies whether the persistent state may be read and written or only read.

```
<!ELEMENT accessmode EMPTY>
<!ATTLIST accessmode
                mode (READ_ONLY|READ_WRITE) #REQUIRED
>
```

The **mode** attribute identifies the access mode.

69.4.5.3 *The catalog Element*

Child element of **containermanagedpersistence**.

The **catalog** element identifies the catalog to used in loading and storing persistent state.

```

<!ELEMENT catalog EMPTY>
<!--ATTLIST catalog
      type CDATA #REQUIRED -->

```

The **type** attribute identifies the type of catalog.

69.4.5.4 *The componentfeatures Element*

Child element of **corbacomponent**.

The **componentfeatures** element is used to describe a component with respect to the components that it inherits from, the interfaces that the component supports, and its provides, uses, emits, publish, and consumes ports. A component also has the features that it inherits from other components. In addition, supported interfaces may inherit from other interfaces. By following the inheritance chain, a graph is formed from the primary component to a set of ports, supported interfaces, and other components. The root component in this graph is identified by the **repositoryid** child element of **corbacomponent**.

The information obtained by traversing the componentfeatures graph may be displayed by graphical tools. But more importantly, it allows component assembly tools to decide what ports on a component are capable of connecting to ports on other components.

```

<!ELEMENT componentfeatures
      ( inheritscomponent?
        , supportsinterface*
        , ports
        , operationpolicies?
        , extension*
        ) >
<!--ATTLIST componentfeatures
      name CDATA #REQUIRED
      repid CDATA #REQUIRED -->

```

The **name** attribute is the non-qualified name of the component.

The **repid** attribute is the fully qualified repository id of the component. **repid** is also used to refer to this component from elsewhere in the descriptor, for example from the **inheritscomponent** element).

69.4.5.5 *The componentkind Element*

Child element of **corbacomponent**.

The **componentkind** element defines the component category. For more information on these categories, see Section 62.1.4, "Component Categories."


```

<!ELEMENT componentkind
( service
| session
| process
| entity
| unclassified
) >

```

69.4.5.6 *The componentproperties Element*

The **componentproperties** element specifies a default component property file. The format of the property file is described in section 69.8 on page 321.

```

<!ELEMENT componentproperties
( fileinarchive
) >

```

69.4.5.7 *The componentrepid Element*

Child element of **corbacomponent**.

componentrepid identifies the repository id of the component described by this descriptor. The repository id also serves to point to the primary **componentfeatures** element for this component within the descriptor, so as to distinguish it from inherited components.

```

<!ELEMENT componentrepid EMPTY >
<!ATTLIST componentrepid
    repid CDATA #IMPLIED >

```

repid is the fully qualified repository id of the component.

69.4.5.8 *The configurationcomplete Element*

Child element of **corbacomponent**.

The **configurationcomplete** attribute is used to set whether `configuration_complete` should be called on the component after it has been fully configured.

```

<!ELEMENT configurationcomplete EMPTY >
<!ATTLIST configurationcomplete
    set ( true | false ) #REQUIRED >

```

69.4.5.9 *The consumes Element*

Child element of **ports**.

A consumes port specifies an event that the component expects to receive. At deployment or creation time, the component will be connected via a channel to other components or entities that emit the event. The **eventpolicy** allows the transaction policy of the event port to be specified.

```
<!ELEMENT consumes
    ( eventpolicy
      , extension* ) >
<!-- consumes
    consumesname CDATA #REQUIRED
    eventtype CDATA #REQUIRED -->
```

consumesname

The **consumesname** attribute identifies the name associated with the consumes statement in idl.

eventtype

The **eventtype** attribute identifies the repository id of the event that the component expects to consume.

69.4.5.10 *The containermanagedpersistence Element*

Child element of **segment**.

An **containermanagedpersistence** element specifies attributes required by the container to manage the component's persistent state using a PSS. **storagehome** indicates the type of abstract storage home, **pssimplementation** identifies a particular PSS implementation to be used, if not specified then the default PSS is used, as determined by the container implementation. **catalog** specifies the catalog type. **accessmode** specifies the access mode--read only or read-write. **psstransactionpolicy** specifies whether transactions are to be used or not and, if so, the isolation level. **params** is used to specify vendor specific parameters.

```
<!ELEMENT containermanagedpersistence
    ( storagehome
      , pssimplementation?
      , catalog?
      , accessmode
      , psstransactionpolicy
      , params?
    ) >
```

69.4.5.11 *The corbacomponent Element*

The root element of this CORBA Component descriptor. See section 69.4.5.1.

69.4.5.12 *The corbaversion Element*

Child element of **corbacomponent**.

The **corbaversion** is used to identify the version of CORBA that the component implementation is assuming. The version is represented by a major and minor number separated by a “.”. For example, “<corbaversion>3.0</corbaversion>”.

<!ELEMENT corbaversion (#PCDATA) >

69.4.5.13 *The emits Element*

Child element of **ports**.

An **emits** port specifies an event that the component generates. At deployment or creation time, the component will be connected to a channel in which it can be connected to consuming components. The **eventpolicy** allows the transaction policy of the event port to be specified.

<!ELEMENT emits

**(eventpolicy
, extension*) >**

<!ATTLIST emits

**emitsname CDATA #REQUIRED
eventtype CDATA #REQUIRED >**

The **emitsname** attribute identifies the name associated with the emits statement in idl.

The **eventtype** attribute identifies the repository id of the emitted events.

69.4.5.14 *The entity Element*

Child element of **componentkind**.

The **entity** component kind is described in Section 62.1.4, “Component Categories,” on page 62-121.

<!ELEMENT entity

(servant) >

69.4.5.15 *The eventpolicy Element*

Child element of **corbacomponent**.

Event policies define the quality of service associated with the event ports of the component. The possible values are defined Section 62.2.8, “Events,” on page 62-128.

```

<!ELEMENT eventpolicy EMPTY>
<!--ATTLIST eventpolicy
      policy ( normal | default | transaction ) #IMPLIED-->

```

69.4.5.16 *The extendedpoapolicy Element*

Child element of **corbacomponent**.

The **extendedpoapolicy** element is a name-value pair used to specify POA policies beyond the base set of policies. It is for new policies, such as firewall, or future POA policies yet to be defined. The **extendedpoapolicy** element must not be used to specify any of the base POA policies. A set of POA policies is predefined for each component category, except for the unclassified category. Only the unclassified component type is flexible with respect to base POA policies; these are set using the **poapolicies** child element of the **unclassified** element.

```

<!ELEMENT extendedpoapolicy EMPTY>
<!--ATTLIST extendedpoapolicy
      name CDATA #REQUIRED
      value CDATA #REQUIRED -->

```

The **name** attribute is the name of the poa policy as defined in the specification where it originated.

The **value** attribute is a valid attribute for the policy as defined in the specification where it originated.

69.4.5.17 *The extension Element*

Child element of **corbacomponent**, **componentfeatures**, **homefeatures**.

See section 69.3.2.11 on page 265.

69.4.5.18 *The fileinarchive Element*

See section 69.3.2.12 on page 265.

69.4.5.19 *The homefeatures Element*

Child element of **corbacomponent**.

The **homefeatures** element is used to describe a component home with respect to the homes that it inherits from and the **operationpolicies** of its interface.

```

<!ELEMENT homefeatures
  ( inheritshome?
    , operationpolicies?
    , extension* ) >
<!ATTLIST homefeatures
  name CDATA #REQUIRED
  repid CDATA #REQUIRED >

```

The **name** attribute is the non-qualified name of the home.

The **repid** attribute is the fully qualified repository id of the home. **repid** is also used to refer to this component from elsewhere in the descriptor, for example from the **inheritshome** element.

69.4.5.20 *The homeproperties Element*

The **homeproperties** element specifies a default home property file. The format of the property file is described in section 69.8 on page 321.

```

<!ELEMENT homeproperties
  ( fileinarchive
  ) >

```

69.4.5.21 *The homerepid Element*

Child element of **corbacomponent**.

homerepid identifies the repository id of the home of the component described by this descriptor. The home repository id also serves to point to the primary **homefeatures** element for the home within the descriptor, so as to distinguish it from inherited homes.

```

<!ELEMENT homerepid EMPTY >
<!ATTLIST homerepid
  repid CDATA #IMPLIED >

```

repid is the fully qualified repository id of the component.

69.4.5.22 *The inheritscomponent Element*

Child element of **componentfeatures**.

The **inheritscomponent** element specifies an inherited component.

```
<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
  repid CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited component, it also serves to refer to the **componentfeatures** element of the inherited component, elsewhere in the descriptor.

69.4.5.23 *The inheritshome Element*

Child element of **homefeatures**.

The **inheritshome** element specifies an inherited home.

```
<!ELEMENT inheritshome EMPTY>
<!ATTLIST inheritshome
  repid CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited home, it also serves to refer to the **homefeatures** element of the inherited home, elsewhere in the descriptor.

69.4.5.24 *The inheritsinterface Element*

Child element of **interface**.

The **inheritsinterface** element is used to specify interface inheritance. This allows, for example, for a derivation chain to be followed from a supported or provided interface up to but excluding the **Object** interface.

```
<!ELEMENT inheritsinterface EMPTY>
<!ATTLIST inheritsinterface
  repid CDATA #REQUIRED>
```

The **repid** identifies is the repository id of the inherited interface, and it is used to refer to the interface element of the inherited interface, elsewhere in the descriptor.

69.4.5.25 *The ins Element*

Child element of **repository**.

The **ins** element is used to specify an interoperable naming service name.

```
<!ELEMENT ins EMPTY>
<!ATTLIST ins
  name CDATA #REQUIRED >
```

name is the INS name.

69.4.5.26 *The interface Element*

Child element of **corbacomponent**.

Specifies an interface that the component, either directly or through inheritance, provides, uses, or supports. The **operationpolicies** child element specifies default transaction policies and required security rights for uses of the interface.

```
<!ELEMENT interface
( inheritsinterface*
, operationpolicies? ) >

<!ATTLIST interface
    name CDATA #REQUIRED
    repid CDATA #REQUIRED >
```

The **name** attribute is the non-qualified name of the interface.

The **repid** attribute is the fully qualified repository id of the interface. **repid** is also used to refer to this interface from elsewhere in the descriptor, for example from the **inheritsinterface** element.

69.4.5.27 *The interop Element*

Child element of **corbacomponent**.

The **interop** element is used to specify whether this component interoperates with another component type by acting as a view for that type or having a view of that type.

```
<!ELEMENT interop EMPTY>
<!ATTLIST interop
    type CDATA #REQUIRED
    direction ( hasview | isview ) #REQUIRED
    descriptor CDATA #REQUIRED >
```

The **type** attribute is the other component type, e.g., “EJB 1.1”.

The **direction** attribute says whether the CORBA component is a view for the other component type or the other way around.

The **descriptor** attribute references the descriptor file of the foreign component within the component archive.

69.4.5.28 *The link Element*

See section 69.3.2.18 on page 268.

69.4.5.29 *The objref Element*

Child element of **repository**.

The **objref** element is used to specify a stringified object reference.

```
<!ELEMENT objref EMPTY>
<!ATTLIST objref
                string CDATA #REQUIRED >
```

The **string** attribute holds the stringified object reference.

69.4.5.30 *The operation Element*

Child element of **operationpolicies**.

The **operation** element is used to specify transaction and required security rights for a particular operation (or group of operations if name="*").

```
<!ELEMENT operation
                ( transaction?
                  , requiredrights? ) >
<!ATTLIST operation
                name CDATA #REQUIRED >
```

The **name** attribute specifies the name of the operation. If the name is specified as "*" then the policies specified by this element apply to all operations in the particular scope in which the **operationpolicies** parent element is defined.

69.4.5.31 *The operationpolicies Element*

Child element of **componentfeatures**, **homefeatures**, **interface**, **provides**, and **supportsinterface**.

The **operationpolicies** element is used to specify a set of operation policies. It consists of a list of operation child elements which each may specify security or transaction policies of an operation or set of operations.

The scope of the **operationpolicies** element depends upon where it is specified. As a child of **componentfeatures** it specifies the policies for the component operations, such as the operations effecting facets, receptacles, and event ports. When used as a child of **homefeatures** it specifies the policies of the home interface operations. As a child of **interface** it specifies the operation policies for all uses of the particular interface. Operation policies set in a **supportsinterface** or **provides** element specify operation policies for a particular use of an interface. Note that operation policies set in **supportsinterface** or **provides** element supersede policies set in an **interface** element.

```
<!ELEMENT operationpolicies
                ( operation+ ) >
```

69.4.5.32 *The param Element*

Child element of **params**.

The **param** element is used to specify a name-value pair.

```
<!ELEMENT param EMPTY >
<!ATTLIST param
    name CDATA #REQUIRED
    value CDATA #REQUIRED >
```

The **name** attribute specifies the name.

The **value** attribute specifies the value.

69.4.5.33 *The params Element*

Child element of **containermanagedpersistence**.

The **params** element is used to specify a set of one or more name-value pairs.

```
<!ELEMENT params (param+) >
```

69.4.5.34 *The poapolicies Element*

Child element of **unclassified**.

The **poapolicies** element is used to identify POA creation parameters for an empty container in which an *unclassified* category component will reside.

```
<!ELEMENT poapolicies EMPTY>
<!ATTLIST poapolicies
    thread (ORB_CTRL_MODEL | SINGLE_THREAD_SAFE ) #REQUIRED
    lifespan (TRANSIENT | PERSISTENT ) #REQUIRED
    iduniqueness (UNIQUE_ID | MULTIPLE_ID) #REQUIRED
    idassignment (USER_ID | SYSTEM_ID) #REQUIRED
    servantretention (RETAIN | NON_RETAIN) #REQUIRED
    requestprocessing (USE_ACTIVE_OBJECT_MAP_ONLY
        |USE_DEFAULT_SERVANT
        |USE_SERVANT_MANAGER) #REQUIRED
    implicitactivation (IMPLICIT_ACTIVATION
        |NON_IMPLICIT_ACTIVATION) #REQUIRED >
```

The **poapolicies** attributes are as defined in the base POA specification.

Note – Not all combinations of POA policies are valid. A good component packaging tool will not permit the user to specify invalid POA policy combinations. If however, an invalid combination of policies is used to configure the empty container, the container/POA should throw an exception.

69.4.5.35 *The ports Element*

Child element of **componentfeatures**.

The **ports** element describes what interfaces a component provides and uses, and what events it emits, publishes, and consumes. Any number of uses, provides, emits, publishes, and consumes elements can be specified in any order.

```
<!ELEMENT ports
  ( uses
  | provides
  | emits
  | publishes
  | consumes
  )* >
```

69.4.5.36 *The process Element*

Child element of **componentkind**.

The **process** component kind is described in Section 62.1.4, “Component Categories.

```
<!ELEMENT process
  ( servant ) >
```

69.4.5.37 *The provides Element*

Child element of **ports**.

The **provides** element specifies an interface that is provided by the component.

The optional **operationpolicies** child element allows transaction policies and required rights to be specified for the provided interface. The policies specified here override any policies specified in the **interface** element, as identified by the **repid**.

```
<!ELEMENT provides
  ( operationpolicies?
  , extension* ) >
<!ATTLIST provides
  providesname CDATA #REQUIRED
  repid       CDATA #REQUIRED
  facettag    CDATA #REQUIRED >
```

The **providesname** is the name given to the provides port in IDL.

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

The **facettag** is the tag for the facet. This attribute is used in combination with the **segmentmember** element, defined in section 69.4.5.47 on page 294, to associate a facet with a segment.

69.4.5.38 *The pssimplementation Element*

Child element of **containermanagedpersistence**.

The **pssimplementation** element identifies a particular vendor's PSS implementation.

```
<!ELEMENT pssimplementation EMPTY>
<!ATTLIST pssimplementation
    id CDATA #REQUIRED >
```

The **id** attribute identifies the particular PSS implementation.

69.4.5.39 *psstransaction Element*

Child element of **containermanagedpersistence**

The **psstransaction** element is used to specify the PSS transactional policies associated with the entity or process component.

```
<!ELEMENT psstransaction (psstransactionisolationlevel?) >
<!ATTLIST psstransaction
    policy (TRANSACTIONAL|NON_TRANSACTIONAL) #REQUIRED >
```

69.4.5.40 *psstransactionisolationlevel Element*

Child element of **psstransaction**.

The **psstransactionisolationlevel** element is used to specify the transaction isolation level when persistent store access is transactional.

```
<!ELEMENT psstransactionisolationlevel EMPTY>
<!ATTLIST psstransactionisolationlevel
    level (
        READ_UNCOMMITTED|READ_COMMITTED|
        REPEATABLE_READ|SERIALIZABLE)
    #REQUIRED >
```

The **level** attribute identifies one of four isolation levels.

69.4.5.41 *The publishes Element*

Child element of **ports**.

A publishes port specifies an event that the component publishes. At deployment or creation time, the component will be connected to a channel by which it can be connected to consuming components. The **eventpolicy** allows the transaction policy of the event port to be specified.

```

<!ELEMENT publishes
    ( eventpolicy
      , extension* ) >

<!ATTLIST publishes
    publishesname CDATA #REQUIRED
    eventtype CDATA #REQUIRED >

```

The **publishesname** attribute identifies the name associated with the emits statement in idl.

The **event_type** attribute identifies the repository id of the published events.

69.4.5.42 *The repository Element*

Child element of **corbacomponent**.

The repository element is used to point to a repository, such as the interface repository.

```

<!ELEMENT repository
    ( ins
      | objref
      | link
    ) >

<!ATTLIST repository
    type CDATA #IMPLIED >

```

The **type** attribute specifies the type of repository. Currently, the only predefined value for **type** is “CORBA Interface Repository”.

69.4.5.43 *requiredrights Element*

Child element of **operation** and **security**.

The **requiredrights** element specifies a list of required rights. When used as a child of **operation**, the rights specified must belong to a rights family specified in the **security** element. When used as a child of **security** the list of rights specify the available rights in the rights family.

```

<!ELEMENT requiredrights
    ( right* ) >

```

69.4.5.44 *right Element*

Child element of **requiredrights**.

The **right** element specifies a particular required right. The right must be a member of the rights family specified by the security element.

```

<!ELEMENT right
  ( description? ) >
<!ATTLIST right
  name CDATA #REQUIRED >

```

The **name** attribute is the name of the required right.

69.4.5.45 *The security Element*

Child element of **corbacomponent**.

The **security** element is an optional child element of **corbacomponent**; it is required whenever rights are assigned to component operations within the descriptor. It specifies the rights family assumed when defining component operation rights. The optional **requiredrights** element may be used to document the rights available in the rights family.

```

<!ELEMENT security
  ( requiredrights? ) >
<!ATTLIST security
  rightsfamily CDATA #REQUIRED >

```

The **rightsfamily** attribute defines the rights family; for example, the “CORBA” rights family.

69.4.5.46 *The segment Element*

Child element of **corbacomponent**.

The **segment** element describes a component segment. It consists of a list of one or more **segmentmember** child elements, indicating the facets that the segment supports, and a **containermanagedpersistence** element indicating that the persistent state of the segment is managed by the container. If the **containermanagedpersistence** element is not present then the persistent state, if any, is managed by the component. Note that the **containermanagedpersistence** element is only employed for **entity** and **process** components.

```

<!ELEMENT segment
  ( segmentmember+
    , containermanagedpersistence?
    , extension*
  ) >
<!ATTLIST segment
  name      CDATA #REQUIRED
  segmenttag CDATA #REQUIRED >

```

name is the name of the segment.

segmenttag is the segment’s tag.

69.4.5.47 *The segmentmember Element*

Child element of **segment**.

The **segmentmember** element specifies a facet that is a member of a segment.

```
<!ELEMENT segmentmember EMPTY>
<!ATTLIST segmentmember
    facettag CDATA #REQUIRED >
```

The **facettag** attribute indicates the member facet's tag. It corresponds to a provided interface with the same facet tag elsewhere in the descriptor. (See the **provides** tag element in section 69.4.5.37 on page 290.)

69.4.5.48 *The servant Element*

Child element of **entity**, **process**, session.

Servant lifetime policies control the lifetime of the servant which implements a component's operations and provide an aid to efficiently manage storage of components within a server process. Servant lifetime policies are fixed for **service** components. Servant lifetime policies must be specified for **session**, **process** and **entity** components and are implemented by the component using APIs provided by the container.

```
<!ELEMENT servant EMPTY >
<!ATTLIST servant
    lifetime (component|method|transaction|container) #REQUIRED >
```

The possible values are defined in Section 62.2.5, "Servant Lifetime Management.

69.4.5.49 *The service Element*

Child element of **componentkind**.

Specifies that the component is of the **service** category. The service component kind is described in Section 62.2.13.1, "The Service Component.

```
<!ELEMENT service EMPTY >
```

69.4.5.50 *The session Element*

Child element of **componentkind**.

Specifies that the component is of the **session** category. The **session** component category is described in Section 62.2.13.2, "The Session Component.

**<!ELEMENT session
(servant) >**

69.4.5.51 *The storagehome Element*

Child element of **segment**.

The **storagehome** element specifies an abstract storage home type.

**<!ELEMENT storagehome EMPTY>
<!ATTLIST storagehome
id CDATA #REQUIRED >**

The **id** attribute specifies the repository id of the abstract storage home.

69.4.5.52 *The simple-link-attributes Entity*

See section 69.3.2.27 on page 271.

69.4.5.53 *The supportsinterface Element*

Child element of **componentfeatures**.

The **supportsinterface** element identifies an interface that the component supports, as defined in IDL.

The optional **operationpolicies** child element allows transaction policies and required rights to be specified for the supported interface. The policies specified here override any policies specified in the **interface** element, as identified by the **repid**.

**<!ELEMENT supportsinterface
(operationpolicies?
, extension*) >
<!ATTLIST supportsinterface
repid CDATA #REQUIRED >**

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

69.4.5.54 *The threading Element*

Child element of **corbacomponent**.

The **threading** element determines the threading policy of the container in which it is placed.

<!ELEMENT threading EMPTY>
<!ATTLIST threading
policy (serialize | multithread) #REQUIRED >

Setting **policy** to **serialize** means that the container will serialize calls to the container.

Setting **policy** to **multithread** means that multiple threads of control can be active in the component at one time.

69.4.5.55 *The transaction Element*

Child element of **corbacomponent**.

The **transaction** element controls the way transactions are managed by the container for this component. Seven possible values can be selected by the component developer to provide maximum flexibility.

<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
use (self-managed|not-supported|required|supports|requires-new|man-
datory|never) #REQUIRED >

If the transaction **use** attribute is set to **self-managed** then it is assumed that the component will manage transactions on its own. Other **use** values indicate that transactions are to be managed by the container; the meaning of these values are defined in the container chapter, Section 62.2.6, “Transactions,” on page 62-126.

69.4.5.56 *The unclassified Element*

Child element of **componentkind**.

The **unclassified** element identifies that the component is of the unclassified sort. See Section 62.2.1, “Component Containers,” on page 62-121 for more information on the unclassified component category.

<!ELEMENT unclassified
(poapolicies) >

69.4.5.57 *The uses Element*

Child element of **ports**.

The **uses** element specifies an interface that is used by the component, as specified in a component IDL *uses* declaration.

<!ELEMENT uses (extension*) >
<!ATTLIST uses
usesname CDATA #REQUIRED
repid CDATA #REQUIRED >

The **usesname** is the name given to the uses port in IDL.

The **repid** is the fully qualified repository id of the component. It is also used to reference an interface element elsewhere in the descriptor.

69.5 Component Assembly Packaging

A component package is the vehicle for deploying a single component implementation, A component assembly package is the vehicle for deploying a set of interrelated component implementations. It is a template or pattern for instantiating a set of components and introducing them to each other.

An assembly package consists of a descriptor and a set of component packages and property files. These files may be packaged together in an archive file or distributed. When distributed, the descriptor represents the package and holds links to its associated files.

The component assembly descriptor describes which components make up the assembly, how those components are partitioned, and how they are connected to each other. A component assembly descriptor is the recipe for deploying a set of interconnected components.

An assembly is normally created visually within a design tool, however it is possible to create assemblies using more primitive tools.

Note – An assembly specifies an *initial* configuration. The actual connected graph of components may evolve beyond that initial configuration. The assembly does not address the evolution of this graph.

69.6 Component Assembly File

The component assembly archive file is a ZIP file containing a component assembly descriptor, a set of component archive files, and, if necessary, a set of component property files. The component assembly archive file has a “.aar” extension.

69.7 Component Assembly Descriptor

A component assembly descriptor is specified using an XML vocabulary. Each component assembly package must contain a single descriptor file. Component descriptors have a “.cad” extension. CAD stands for Component Assembly Descriptor.

The assembly descriptor describes a component assembly. It consists of elements describing the components used in the assembly, connection information, and partitioning information.

A component instantiation is always relative to a home. A deployed home is called a home “placement”.

Component instantiations are connected by their *provides* and *uses* interfaces, or by their *emits*, *publishes*, and *consumes* events. If one component provides an interface of a particular type and another component uses an interface of that type, then we can pass the reference of the provided interface to the component that uses it, in effect connecting the two components. In the same way, we connect two components where one emits or publishes an event that the other consumes.

Sets of component instances may be partitioned. Components may be free or partitioned to a generic set of hosts and processes. This is really a process of conveying that specific components are to be collocated within a single process or host. Free components, components that are not used in a collocation may be deployed in any manner at deployment time.

When used in an archive, the CAD file for the archive is placed in a top level directory called "meta-inf".

69.7.1 Component Assembly Descriptor Example

The following example illustrates how to write a component assembly descriptor. For further information, see the element descriptions that follow and the XML DTDs in the appendix.

```
<!DOCTYPE componentassembly SYSTEM "componentassembly.dtd">
```

```
<componentassembly id="ZZZ123">
  <description>Example assembly</description>
  <componentfiles>
    <componentfile id="A">
      <fileinarchive name="ca.ccd"/>
    </componentfile>
    <componentfile id="B">
      <fileinarchive name="cb.ccd"/>
    </componentfile>
    <componentfile id="C">
      <fileinarchive name="cc.ccd">
        <link href="ftp://www.xyz.com/car/cc.car"/>
      </fileinarchive>
    </componentfile>
    <componentfile id="D">
      <fileinarchive name="cd.ccd"/>
    </componentfile>
    <componentfile id="E">
      <fileinarchive name="ce.ccd"/>
    </componentfile>
    <componentfile id="F">
      <fileinarchive name="cf.ccd"/>
    </componentfile>
  </componentfiles>

  <partitioning>

    <homeplacement id="AaHome">
      <componentfileref idref="A"/>
      <componentinstantiation id="Aa"/>
    </homeplacement>

    <processcollocation cardinality="*">
      <usagename>Example process collocation</usagename>
      <impltype language="C++" /> <!-- optional -->
      <homeplacement id="BbHome">
        <componentfileref idref="B"/>
        <componentinstantiation id="Bb"/>
      </homeplacement>
      <homeplacement id="CcHome">
        <componentfileref idref="C"/>
        <componentinstantiation id="Cc"/>
      </homeplacement>
    </processcollocation>
  </partitioning>
</componentassembly>
```

```

    </homeplacement>
</processcollocation>

<hostcollocation cardinality="1">
  <usagename>Example host collocation</usagename>
  <processcollocation cardinality="*">
    <homeplacement id="DdHome">
      <componentfileref idref="D"/>
      <componentinstantiation id="Dd"/>
    </homeplacement>
    <homeplacement id="EdHome">
      <componentfileref idref="E"/>
      <componentinstantiation id="Ee"/>
    </homeplacement>
  </processcollocation>
  <homeplacement id="FfHome">
    <componentfileref idref="F"/>
    <componentinstantiation id="Ff"/>
  </homeplacement>
</hostcollocation>

<homeplacement id="AaaHome">
  <usagename>Example home for A components</usagename>
  <componentfileref idref="A"/>
  <componentimplref idref="an A impl"/>
  <homeproperties>
    <fileinarchive name="AHomeProperties.cpf"/>
  </homeproperties>
  <componentproperties>
    <fileinarchive name="defaultAProperties.cpf"/>
  </componentproperties>
  <registerwithhomefinder name="AaHome"/>

  <componentinstantiation id="Aaa">
    <usagename>Example component instantiation </usagename>
    <componentproperties>
      <fileinarchive name="AaaProperties.cpf"/>
    </componentproperties>
    <registercomponent>
      <registerwithnaming name="sink"/>
      <registerwithtrader>
        <traderproperties>
          <traderproperty>
            <traderpropertyname>ppm</traderpropertyname>
            <traderpropertyvalue>10</traderpropertyvalue>
          </traderproperty>
          <traderproperty>
            <traderpropertyname>weight</traderpropertyname>
            <traderpropertyvalue>333</traderpropertyvalue>
          </traderproperty>
        </traderproperties>
      </registerwithtrader>
    </registercomponent>
  </componentinstantiation>

```

```

        </registerwithtrader>
        </registercomponent>
    </componentinstantiation>
</homeplacement>

</partitioning>

<connections>
    <connectinterface>
        <usesport>
            <usesidentifier>abc</usesidentifier>
            <componentinstantiationref idref="Aa"/>
        </usesport>
        <providesport>
            <providesidentifier>abc</providesidentifier>
            <componentinstantiationref idref="Bb"/>
        </providesport>
    </connectinterface>
    <connectevent>
        <consumesport>
            <consumesidentifier>pqr</consumesidentifier>
            <componentinstantiationref idref="Aaa"/>
        </consumesport>
        <emitsport>
            <emitsidentifier>mno</emitsidentifier>
            <componentinstantiationref idref="Ee"/>
        </emitsport>
    </connectevent>
</connections>

</componentassembly>

```

69.7.2 Component Assembly Descriptor XML Elements

This section describes the XML elements that make up a component assembly descriptor. The section is organized starting with the root element of the descriptor document, **componentassembly**, followed by all subordinate elements, in alphabetical order. The complete component assembly DTD may be found in Section 695.4, “componentassembly.dtd,” on page 695-346.

69.7.2.1 The componentassembly Root Element

The **componentassembly** element is the root element of the component assembly descriptor. The **description** element is text describing the assembly. The **componentfiles** element lists the component files that are used in the assembly, the **partitioning** element describes how homes and components are to be deployed. The

connections element describes how deployed components and homes are to be connected. The **extension** element can be used to add proprietary or experimental elements to the component assembly document.

```
<!ELEMENT componentassembly
    ( description?
      , componentfiles
      , partitioning
      , connections?
      , extension*
    ) >
<!ATTLIST componentassembly
    id ID #REQUIRED
    derivedfrom CDATA #IMPLIED >
```

The **id** attribute is a DCE UUID which uniquely identifies the assembly.

The **derivedfrom** attribute is used to point to an assembly from which this assembly was derived. The **derivedfrom** attribute contains the id of the source assembly.

Note – The **derivedfrom** attribute is for a deployment tool that wants to create a copy of an assembly descriptor and archive to describe an actual deployment; it maintains the relationship between the “clone” and the original. The new assembly descriptor would have the destination addresses for each placement and collocation defined; and collocations with non-ordinal cardinality in the original assembly would be copied to one or more collocations, with singular cardinality, in the derived assembly. The new archive file might prune constituent component archive files to contain single implementations to facilitate copying component implementations to target deployment hosts.

69.7.2.2 *The codebase Element*

See section 69.3.2.4 on page 262.

69.7.2.3 *The componentfile Element*

The **componentfile** element refers to a component archive file containing a component and home implementation. **componentfile** elements are referenced by **homeplacement** elements.

componentfile contains either a **fileinarchive**, **link** or **codebase** element.

```

<!ELEMENT componentfile
    ( fileinarchive
    | codebase
    | link
    ) >
<!-- ATTLIST componentfile
    id ID #REQUIRED
    type CDATA #IMPLIED -->

```

The **id** attribute must uniquely identify the **componentfile** element within the descriptor.

The optional **type** attribute specifies the type of component file. If unspecified then the file is assumed to be CORBA component. An example use of the type attribute would be to specify an EJB component file, where **type**="EJB 1.1".

69.7.2.4 *The componentfileref Element*

The **componentfileref** element refers to a particular **componentfile** element in the **componentfiles** block.

```

<!ELEMENT componentfileref EMPTY >
<!-- ATTLIST componentfileref
    idref CDATA #REQUIRED -->

```

The **idref** attribute corresponds to a unique **componentfile id** attribute.

69.7.2.5 *The componentfiles Element*

The **componentfiles** element is used to list all of the component files that are used in the assembly. At least one component file must be specified.

Each component file is uniquely identified for reference elsewhere in the descriptor. Multiple component instances may refer to a single component file.

```

<!ELEMENT componentfiles
    ( componentfile+
    ) >

```

69.7.2.6 *The componentimplref Element*

The **componentimplref** element is used to refer to a particular implementation in a component file.


```

<!ELEMENT componentimplref EMPTY >
<!ATTLIST componentimplref
    idref CDATA #REQUIRED >

```

The **idref** attribute refers to a unique **implementation** element **id** in the component descriptor. The **componentimplref** is optional if there is only one implementation in the component file. Or it may be set at deployment time depending on the type of platform that the component is deployed to.

69.7.2.7 *The componentinstantiation Element*

The **componentinstantiation** element describes a particular instantiation of a component relative to a home placement. The **componentinstantiation** element is a direct child of the **homeplacement** element.

The **usagename** child element is used to specify a name for the placement, possibly for display in a tool. The **componentproperties** element refers to a property file associated with this instantiation. It is used to configure the component once it is created and after the home sets initial property values (as specified in the **homeplacement componentproperties** element). The **registercomponent** element instructs the installation process to register the component or its provided interfaces with a naming service or trader.

```

<!ELEMENT componentinstantiation
    ( usagename?
      , componentproperties?
      , registercomponent*
      , extension*
    ) >
<!ATTLIST componentinstantiation
    id ID #REQUIRED >

```

The **id** attribute is a unique identifier within the assembly descriptor for the component. The **id** is used to refer to the component instance in the connect block.

69.7.2.8 *The componentinstantiationref Element*

The **componentinstantiationref** element refers to a particular **componentinstantiation** element in the assembly descriptor.

```

<!ELEMENT componentinstantiationref EMPTY >
<!ATTLIST componentinstantiationref
    idref CDATA #REQUIRED >

```

The **idref** attribute corresponds to a unique **componentinstantiation** **id** attribute.

69.7.2.9 *The componentproperties Element*

The **componentproperties** element specifies a property file for a home. If the component file has a default property file in the component package, the component property file overrides the default. The property file may be specified by either a **fileinarchive** or a **codebase** child element. The format of the property file is described in section 69.8 on page 321.

When the **componentproperties** element is specified as part of a **homeplacement** element, then the properties are used to configure each component created through that home. When **componentproperties** is specified as part of a **componentinstantiation** element, the properties are used to configure that single instantiation. If component properties are set on both a **homeplacement** and an associated **componentinstantiation**, then the component will be configured first by the **homeplacement** component properties and then by the **componentinstantiation** component properties.

```
<!ELEMENT componentproperties
    ( fileinarchive
    | codebase
    ) >
```

69.7.2.10 *The componentsupportedinterface Element*

Specifies a component with a *supports* interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element. The component is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT componentsupportedinterface
    ( componentinstantiationref
    | findBy
    )>
```

69.7.2.11 *The connectevent Element*

The **connectevent** element is used in the **connections** element to specify a connection from a *consumes* port, of one component, to an *emits* or *publishes* port of another component.

The **consumesport** element identifies a component and associated consumes port. The **emitsport** element identifies a component associated emits port. The **publishesport** element identifies a component and associated **publishesport**.

```

<!ELEMENT connectevent
    ( consumesport
      , ( emitsport
        | publishesport
        )
      ) >
<!ATTLIST connectevent
    id ID #IMPLIED >

```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor, however someone (or a tool) might want to use it to refer to a particular **connectevent** element.

69.7.2.12 *The connecthomes Element*

The **connecthomes** element is used to specify a connection between a **proxyhome** and another home.

The **proxyhome** element refers to the proxy home. The **destinationhome** element refers to the home to which the proxy home will be connected. The destination home can be either another proxy home or an actual home.

```

<!ELEMENT connecthomes
    ( proxyhome
      , destinationhome
      ) >
<!ATTLIST connecthomes
    id ID #IMPLIED >

```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor, however someone (or a tool) might want to use it to refer to a particular **connecthome** element.

69.7.2.13 *The connectinterface Element*

The **connectinterface** element is used to connect a component's *uses* port to an interface. The interface may be a provided or supported interface of another component, it may be an existing interface (other than those provided by components in the assembly), or it may be a home interface.

The **usesport** element identifies the component and port where the connection is to be made. The **providesport** element identifies a component and provides port. The **componentsupportedinterface** element identifies a component that has a supported interface which will satisfy the uses port. The **existinginterface** element identifies a way to find an existing interface that will satisfy the *uses*. The **homeinterface** element identifies a **homeinterface** that the uses port requires.

```

<!ELEMENT connectinterface
  ( usesport
    , ( providesport
        | componentsupportedinterface
        | existinginterface
        | homeinterface
      )
    ) >
<!ATTLIST connectinterface
  id ID #IMPLIED >

```

The **id** attribute is a unique identifier within the assembly descriptor. It is not required or used elsewhere in the assembly descriptor, however someone (or a tool) might want to use it to refer to a particular **connectinterface** element.

69.7.2.14 *The connections Element*

The connections element is used to satisfy component uses and consumes dependencies and to connect homes. The **connectinterface** element is used to connect component *uses* ports to interfaces. the **connectevent** element is used to connect a components *consumes* port to event producers. The **connecthome** element is used to connect a proxy home to another home.

```

<!ELEMENT connections
  ( connectinterface
    | connectevent
    | connecthome
    | extension
  )* >

```

Note – If a **componentinstantiation** involved in a connection has a cardinality greater than 1, or if it is part of a process or host collocation with a cardinality greater than 1, then multiple connections will be realized from or to each instance of the component. That is, the connection will be made for each instantiation of the component.

69.7.2.15 *The consumesidentifier Element*

A child element of **consumingcomponent**, **consumesidentifier** identifies which *consumes* “port” on the component is to participate in the relationship. The type of the consumes event must match the type of the connected emits or publishes event.

<!ELEMENT consumesidentifier (#PCDATA) >

69.7.2.16 *The consumesport Element*

Specifies the event-consuming side of an event connection relationship. The **consumesidentifier** child element identifies the particular *consumes* port. The component with this consumes port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```
<!ELEMENT consumesport
      ( consumesidentifier
        , ( componentinstantiationref
          | findBy
          )
        )
      >
```

69.7.2.17 *The description Element*

The **description** element contains a string description. It is used to describe its parent element. It contains string content.

<!ELEMENT description (#PCDATA) >

69.7.2.18 *The destination Element*

The **destination** element is used to record where a **homeplacement**, **executableplacement**, **hostcollocation**, or **processcollocation** is to be (or has been) deployed. The format of the destination string is determined by a particular deployment tool.

<!ELEMENT destination (#PCDATA) >

69.7.2.19 *The destinationhome Element*

Identifies a home to be connected to by a proxy home. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, naming service, or trader, or using a stringified object reference.

```

<!ELEMENT destinationhome
    ( homeplacementref
    | findby
    ) >

```

69.7.2.20 *The emitsidentifier Element*

The **emitsidentifier** identifies an emits “port” on a component. The identifier corresponds to a emits identifier specified in IDL.

```

<!ELEMENT emitsidentifier ( #PCDATA ) >

```

69.7.2.21 *The emitsport Element*

Specifies the event-emitting side of an event connection relationship. The **emitsidentifier** child element identifies the particular *emits* port. The component with this emits port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```

<!ELEMENT emitsport
    ( emitsidentifier
    , ( componentinstantiationref
    | findby
    )
    ) >

```

69.7.2.22 *The executableplacement Element*

This **executableplacement** element describes a deployment of an executable. The **executableplacement** element may be a direct child of the **partitioning** element which states that it has no collocation constraints; or it may be a child element of the **hostcollocation** element.

The **usagename** child element is used to specify a name for the placement, possibly for use in a tool. The **componentfileref** element specifies the component file. The **componentimplref** element refers to a specific implementation in the **softpkg** descriptor. Note that the implementation referred to by **componentimplref** must have a code type of “Executable”. The **invocation** element specifies any arguments with which the executable should be invoked. The **destination** element is used to record where the **executableplacement** is to be deployed.

```

<!ELEMENT executableplacement
    ( usagename?
      , componentfileref
      , componentimplref
      , invocation?
      , destination?
      , extension*
    ) >
<!ATTLIST executableplacement
    id          ID #REQUIRED
    cardinality CDATA "1" >

```

The **id** attribute is a unique identifier within the assembly descriptor for the **executableplacement**.

The **cardinality** attribute specifies how many instantiations of this executable may be deployed. Possible values for cardinality are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

69.7.2.23 *The existinginterface Element*

Specifies an interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element. The **findby** element points to an existing interface that can be found within a naming service or trader, or using a stringified object reference.

```

<!ELEMENT existinginterface
    ( findBy ) >

```

69.7.2.24 *The extension Element*

See section 69.3.2.11 on page 265.

69.7.2.25 *The fileinarchive Element*

See section 69.3.2.11 on page 265.

69.7.2.26 *The findby Element*

The **findby** element is used to resolve a connection between two components. It tells the installation agent how to locate a party, usually a component, interface, or home, involved in the relationship. In the simplest case, the installer will know where the item is because it was the one responsible for installing it. But if the item to be located already exists in the installation environment, the installer must know how to find it. It could locate a component in a naming service, in a trader, a home finder, or by a stringified object reference. The purpose of the **findby** element is to provide such information.

The **namingservice** element specifies a naming service name. The **stringifiedobjectref** element is a stringified IOR for the item. The **traderquery** is a query for locating the item in a trader. The **homefinder** is a name to look up a home in a home finder.

```
<!ELEMENT findby
    ( namingservice
    | stringifiedobjectref
    | traderquery
    | homefinder
    | extension
    ) >
```

69.7.2.27 *The homefinder Element*

The **homefinder** element is used to indicate a home finder name for a home.

```
<!ELEMENT homefinder EMPTY >
<!ATTLIST homefinder
    name CDATA #REQUIRED >
```

The **name** attribute specifies the name of the home as registered with the home finder. Home finders are defined in Section 61.8, “Home Finders.”

69.7.2.28 *The homeinterface Element*

Specifies a home with an interface that can satisfy an interface connection to a *uses* port within a **connectinterface** element. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, a naming service or trader, or using a stringified object reference.

```
<!ELEMENT homeinterface
    ( homeplacementref
    | findby
    ) >
```

69.7.2.29 *The homeplacement Element*

This **homeplacement** element describes a particular deployment of a component home. The **homeplacement** element may be a direct child of the **partitioning** element which states that it has no collocation constraints; or it may be a child element of the **hostcollocation** or **processcollocation** elements which states specific host or process collocation constraints.

The **usagename** child element is used to specify a name for the placement, possibly for use in a tool. The **componentfileref** element specifies the component file. The **componentimplref** element refers to a specific implementation in the component file. The **homeproperties** element refers to a state file associated with the home placement; it is used to configure the home after it is created. The **componentproperties** element refers to a property file used to configure all components created through the home.

The **registerwithhomefinder** element instructs the installation process to register the home with the home finder. The **registerwithnaming** element instructs the installation process to register the home with a naming service. The **registerwithtrader** element instructs the installation process to register the home with a trader service. The **componentinstantiation** element instructs the installation agent to create a component using this home. The **destination** element is used to record where the **homeplacement** is to be deployed, if designated.

<!ELEMENT homeplacement

```
( usagename?
, componentfileref
, componentimplref?
, homeproperties?
, componentproperties?
, registerwithhomefinder*
, registerwithnaming*
, registerwithtrader*
, componentinstantiation*
, destination?
, extension*
) >
```

<!ATTLIST homeplacement

```
id ID #REQUIRED
cardinality CDATA "1" >
```

The **id** attribute is a unique identifier within the assembly descriptor for the **homeplacement**. The **id** is used to refer to the home in the connect block.

The **cardinality** attribute specifies how many instantiations of this component may be deployed. Possible values for cardinality are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1 and there are any connections to this **homeplacement**, then connections will be made to each instance of the deployed home.

69.7.2.30 *The homeplacementref Element*

The **homeplacementref** element refers to a particular **homeplacement** element in the assembly descriptor.

<!ELEMENT homeplacementref EMPTY >

<!ATTLIST homeplacementref
idref CDATA #REQUIRED >

The **idref** attribute corresponds to a unique **homeplacement** id attribute.

69.7.2.31 *The homeproperties Element*

The **homeproperties** element specifies a property file for a home. The properties are used to configure the home when it is created. The property file may be specified by either a **fileinarchive** or a **codebase** child element. The format of the property file is described in section 69.8 on page 321.

```
<!ELEMENT homeproperties
    ( fileinarchive
    | codebase
    ) >
```

69.7.2.32 *The hostcollocation Element*

A **hostcollocation** specifies a group of component instances that are to be deployed together to a single host. The child elements are an optional **usagename**, an optional **impltype**, and a list of **processcollocation**, **homeplacement**, and **executableplacement** elements. If **impltype** is specified then each of the component instances must have implementations supporting the implementation type. If **impltype** is not specified, then at deployment time each of the collocated components must have implementations supporting the target deployment platform.

```
<!ELEMENT hostcollocation
    ( usagename?
    , impltype?
    , ( homeplacement
    | executableplacement
    | processcollocation
    | extension
    )+
    , destination?
    ) >
<!ATTLIST hostcollocation
    id ID #IMPLIED
    cardinality CDATA "1" >
```

The **id** attribute uniquely identifies this host collocation in the component assembly file. The **cardinality** attribute specifies how many instances of this host collocation may be deployed. Possible values for **cardinality** are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1, and there are connections to components within the **hostcollocation**, then connections will be made to the corresponding components or component homes within each instance of the collocation.

69.7.2.33 *The impltype Element*

Issue – May not be necessary.

69.7.2.34 *The invocation Element*

The **invocation** element is used to specify invocation arguments for an executable placement.

```
<!ELEMENT invocation EMPTY >
<!ATTLIST invocation
    args CDATA #REQUIRED >
```

The **args** attribute is a string containing the arguments to be used in invoking the executable. Note, that **args** is just the arguments to the executable, it does not include the executable name.

69.7.2.35 *The link Element*

See section 69.3.2.18 on page 268.

69.7.2.36 *The namingservice Element*

The naming service element is used to indicate that a component or interface should be found using a naming service.

```
<!ELEMENT namingservice EMPTY >
<!ATTLIST namingservice
    name CDATA #REQUIRED >
```

The name attribute specifies the naming service name to look up.

69.7.2.37 *The partitioning Element*

Component partitioning specifies a deployment pattern of homes and components to generic processes and hosts. The pattern is expressed via collocation constraints.

A particular usage of a component is always relative to a component home. Uses of component homes are recognized in the assembly as home placements. A home placement, and component instantiations relative to that home, may be collocated with other home placements and component instantiations in a process. Processes and home placements may be collocated within a logical host. A home placement that is not part of a process or host collocation may be deployed without constraint.

An executable placement is the placement of a particular executable. It may be partitioned without constraint or as part of a host collocation.

Within a **partitioning** element, **homeplacement**, **executableplacement** and collocation constraints are specified. The **homeplacement** child element specifies a freely deployable home. The **executableplacement** element specifies a freely deployable executable. The **processcollocation** and **hostcollocation** child elements are used to group **homeplacement** together into deployable units.

A **homeplacement** may be declared as part of a host or process collocation or by itself. The actual host and process will be determined at deployment time. Home placements, executable placements, process collocations, and host collocations all have an associated cardinality. The default cardinality is “1”. An ordinal cardinality of 1 or greater mandates that the deployable unit must be instantiated that many times, cardinality of “+” indicates 1 or more, and “*” indicates zero or more.

```
<!ELEMENT partitioning
( homeplacement
| executableplacement
| processcollocation
| hostcollocation
| extension
)* >
```

69.7.2.38 *The processcollocation Element*

The **processcollocation** element specifies a group of home and associated component instantiations that are to be deployed together to a single process. The child elements are an optional **usagename**, an optional **impltype**, and a list of **homeplacement** elements. If **impltype** is specified then each of the component instances must have implementations supporting the implementation type. If **impltype** is not specified, then at deployment time each of the collocated components have implementations supporting the target deployment platform.

```
<!ELEMENT processcollocation
( usagename?
, impltype?
, ( homeplacement
| extension
)+
) >
<!ATTLIST processcollocation
id ID #IMPLIED
cardinality CDATA "1" >
```

The **id** attribute uniquely identifies this process collocation in the component assembly file. The **cardinality** attribute specifies how many instances of this process collocation may be deployed. Possible values for **cardinality** are a specific number, a “+” to specify 1 or more, or a “*” to specify 0 or more. The default cardinality is “1”.

Note that if the **cardinality** is greater than 1, and there are connections to components and homes within the **processcollocation**, then connections will be made to corresponding components or component homes within each instance of the collocation.

69.7.2.39 *The providesidentifier Element*

The **providesidentifier** identifies a provides “port” on a component. The identifier corresponds to a provides identifier specified in component IDL.

<!ELEMENT providesidentifier (#PCDATA) >

69.7.2.40 *The providesport Element*

Specifies the interface providing side of an interface connection relationship. The **providesidentifier** child element identifies the particular *provides* port. The component with this provides port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

<!ELEMENT providesport
 (providesidentifier
 , (componentinstantiationref
 | findBy
)
)>

69.7.2.41 *The publishesidentifier Element*

The **publishesidentifier** identifies a publishes “port” on a component. The identifier corresponds to the identifier specified in IDL for the publishes port.

<!ELEMENT publishesidentifier (#PCDATA) >

69.7.2.42 *The publishesport Element*

Specifies the event-publishes side of an event connection relationship. The **publishesidentifier** child element identifies the particular *publishes* port. The component with this publishes port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

```

<!ELEMENT publishesport
    ( publishesidentifier
      , ( componentinstantiationref
        | findby
        )
      )+
    )>

```

69.7.2.43 *The registercomponent Element*

The **registercomponent** element is used to specify that a component, a provided interface, or a published event should be registered with a naming service or trader.

Issue – In the case of events, what gets registered?

If an **emitsidentifier**, **providesidentifier**, or **publishesidentifier** is specified then that element is registered. If none of the above are specified then it is implied that the component itself is to be registered.

Registration may be through a naming service or trader. The **registerwithnaming** element specifies a naming service registration and **registerwithtrader** specifies a trader registration. The interface, event, or component registration may be registered with both a naming service and a trader, multiple times. At least one registration must take place.

```

<!ELEMENT registercomponent
    ( ( emitsidentifier
      | providesidentifier
      | publishesidentifier
      )?
      , ( registerwithnaming
        | registerwithtrader
        )+
    )>

```

69.7.2.44 *The registerwithhomefinder Element*

The **registerwithhomefinder** element tells the installer to register a component home with the home finder.

```

<!ELEMENT registerwithhomefinder EMPTY >
<!ATTLIST registerwithhomefinder
    name CDATA #REQUIRED >

```

The **name** attribute is the name to register the home with in the home finder.

69.7.2.45 *The registerwithnaming Element*

The **registerwithnaming** element tells the installer to register a component instance or home with a naming service after it is created.

```

<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming
              name CDATA #IMPLIED >

```

The **name** attribute is the naming service name. If the name is not specified, it will be determined at deployment time, possibly with interaction with the user.

69.7.2.46 *The registerwithtrader Element*

The **registerwithtrader** element tells the installer to register a component instance or home with a trader after it is created.

```

<!ELEMENT registerwithtrader
              ( traderproperties ) >
<!ATTLIST registerwithtrader
              tradername CDATA #IMPLIED >

```

69.7.2.47 *The proxyhome Element*

Identifies a proxy home that is to be connected to another home. The home is identified by a **homeplacementref** or a **findby** element. The **homeplacementref** identifies a home within the assembly. The **findby** element points to an existing home that can be found within a home finder, naming service, or trader, or using a stringified object reference.

```

<!ELEMENT remotehome
              ( homeplacementref
                | findBy
              ) >

```

69.7.2.48 *The stringifiedobjectref Element*

The stringifiedobjectref element is used to locate a component by its object reference.

```

<!ELEMENT stringifiedobjectref ( #PCDATA ) >

```

69.7.2.49 *Trader elements*

The trader elements are used to register a home, component or interface with a trader and to find a home, component or interface using a trader query. The trader elements closely parallel trader functionality in name and purpose.

Issue – The trader elements have to be reviewed to make sure that they serve the purpose intended. Also, consider using a property file.

```
<!ELEMENT traderconstraint ( #PCDATA ) >

<!ELEMENT traderexport
  ( traderservicetypename
    , traderproperties
  ) >

<!ELEMENT traderpolicy
  ( traderpolicyname
    , traderpolicyvalue
  ) >

<!ELEMENT traderpolicyname ( #PCDATA ) >

<!ELEMENT traderpolicyvalue ( #PCDATA ) >

<!ELEMENT traderpreference ( #PCDATA ) >

<!ELEMENT traderproperties
  ( traderproperty+ ) >

<!ELEMENT traderproperty
  ( traderpropertyname
    , traderpropertyvalue
  ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ( #PCDATA ) >

<!ELEMENT traderquery
  ( traderservicetypename
    , traderconstraint
    , traderpreference?
    , traderpolicy*
    , traderspecifiedprop*
  ) >

<!ELEMENT traderservicetypename ( #PCDATA ) >

<!ELEMENT traderspecifiedprop ( #PCDATA ) >
```

Note – These still need to be explained in text. In the mean time, look at the trader spec. The correspondence should be obvious.

69.7.2.50 *The usagename Element*

A user defined “friendly” name.

<!ELEMENT usagename (#PCDATA) >

69.7.2.51 *The usesidentifier Element*

A child element of **usingcomponent**, **usesidentifier** identifies which uses “port” on the component is to participate in the relationship. The type of the using interface must match the type of the connected provides interface.

<!ELEMENT usesidentifier (#PCDATA) >

69.7.2.52 *The usingcomponent Element*

Specifies the interface using side of an interface connection relationship. The **usesidentifier** child element identifies the particular *uses* port. The component with this uses port is identified by a **componentinstantiationref** or a **findby** element. The **componentinstantiationref** identifies a component within the assembly. The **findby** element points to an existing component that can be found within a naming service or trader, or using a stringified object reference.

<!ELEMENT usesport

```
( usesidentifier
  , ( componentinstantiationref
    | findBy
    )
  )
>
```

69.8 *Property File Descriptor*

The property file details component or home attribute settings. Properties are described using an XML vocabulary described below. The property file is used at deployment time to configure a home or component instance. A configurator uses the property file to determine how to set component and component home property attributes.

The property file may be edited using a text editor or with the help of a GUI tool. A packaged component may be shipped with a set of default properties that may be altered by the end user.

The suggested file extension for property files is “.cpf”, for Component Property File.

69.8.1 *Property File Example*

The following property descriptor example has 3 properties: **bufferSize**, **niceGuys**, and **sanityTestTime**. The **bufferSize** parameter is a long type; the **niceGuys** property is a sequence of strings; and the **sanityTestTime** property is a structure of type **timestruct**, containing 3 shorts.

```

<properties>
  <simple name=bufSize type="long">
    <description>Size of Whizitron input buffer</description>
    <value>4096</value>
    <defaultvalue>256</defaultvalue>
  </simple>
  <sequence name="niceGuys" type="sequence<string>">
    <simple type="string"><value>Dave</value></simple>
    <simple type="string"><value>Ed</value></simple>
    <simple type="string"><value>Garrett</value></simple>
    <simple type="string"><value>Jeff</value></simple>
    <simple type="string"><value>Jim</value></simple>
    <simple type="string"><value>Martin</value></simple>
    <simple type="string"><value>Patrick</value></simple>
  </sequence>

  <struct name="sanityTestTime" type="timestruct">
    <description>Time to start daily sanity check</description>
    <simple name="hour" type="short"><value> 24 </value></simple>
    <simple name="minute" type="short"><value> 0 </value></simple>
    <simple name="second" type="short"><value> 0 </value></simple>
  </struct>
</properties>

```

The **properties** document has 3 major elements: **simple**, **sequence** and **struct**.

The **simple** element describes a single primitive idl type. The **sequence** element corresponds to an IDL sequence, and the **struct** element corresponds to an IDL struct.

Note – If the user of the property file does not have static information about the types specified in the property file then it will likely need to construct the type into a **DynAny**.

69.8.2 Property File XML Elements

This section describes the XML elements that make up a properties file. The section is organized starting with the root element of the properties document, **properties**, followed by all subordinate elements, in alphabetical order. The complete properties file DTD may be found in Section 695.3, “properties.dtd,” on page 695-345.

69.8.2.1 *The properties Root Element*

The **properties** element is the root element of the properties document. It contains an optional description and any combination of **simple**, **sequence**, and **struct** elements.

```
<!ELEMENT properties
    ( description?
      , ( simple
        | sequence
        | struct
        )*
    ) >
```

69.8.2.2 *The choice Element*

```
<!ELEMENT choice ( #PCDATA ) >
```

The **choice** element is used to specify a valid simple property value.

69.8.2.3 *The choices Element*

```
<!ELEMENT choices ( choice+ ) >
```

The **choices** element is a list of one or more choice elements.

69.8.2.4 *The defaultvalue Element*

```
<!ELEMENT defaultvalue ( #PCDATA ) >
```

The **defaultvalue** element is used to specify a default simple property value.

69.8.2.5 *The description Element*

```
<!ELEMENT description ( #PCDATA ) >
```

The **description** element is used to provide a description of its enclosing element.

69.8.2.6 *The properties Element*

The root element of the properties file. See section 69.8.2.1 on page 323.

69.8.2.7 The simple Element

The **simple** element is used to specify an attribute value of a primitive type. **simple** contains a mandatory **value** element, and optional **description**, **choices**, and **defaultvalue** elements.

The **value** element is used to specify the value of the simple type. If the **value** element is empty, the value is deemed unspecified. If the value is unspecified, and there is a **defaultvalue** defined, then the default value will be used.

The **description**, **choices** and **defaultvalue** child elements may be used to provide guidance to the end user in deciding how to set the attributes.

<!ELEMENT simple

```
( description?  
  , value  
  , choices?  
  , defaultvalue?  
 ) >
```

<!ATTLIST simple

```
  name CDATA #IMPLIED  
  type (  boolean  
         | char  
         | double  
         | float  
         | short  
         | long  
         | objref  
         | octet  
         | short  
         | string  
         | ulong  
         | ushort  
        ) #REQUIRED >
```

name

The **name** attribute specifies the name of the attribute as it appears in IDL. The name attribute is required, except when the property is used in a sequence.

type

The **type** attribute specifies the type of the corresponding attribute. Property types are either an IDL primitive data type, or an objref.

Note – The objref is in its stringified form in the property element. The stringified object reference is converted into a proper object reference before being assigned to its corresponding attribute.

69.8.2.8 *The sequence Element*

The **sequence** element is used to represent a sequence of similar types. It may be a sequence of simple types, a sequence of structs, or a sequence of sequences. The order of the sequence elements in the property file is preserved in the constructed sequence. An optional description may be used to describe the sequence property.

```
<!ELEMENT sequence
    ( description?
      , ( simple*
        | struct*
        | sequence*
        )
    ) >

<!ATTLIST sequence
    name CDATA #IMPLIED
    type CDATA #REQUIRED >
```

name

The **name** attribute specifies the name of the sequence as it appears in IDL. The name attribute is required, except when the sequence property is used in another sequence.

type

The **type** attribute specifies the type of the corresponding IDL sequence. The type of each element in the sequence must match the sequence type.

69.8.2.9 *The struct Element*

The **struct** element corresponds to an IDL structure. It may be composed of simple properties, sequences, or other structs.

```
<!ELEMENT struct
    ( description?
      , ( simple
        | sequence
        | struct
        )*
    ) >

<!ATTLIST struct
    name CDATA #IMPLIED
    type CDATA #REQUIRED >
```

name

The **name** attribute specifies the name of the struct attribute as it appears in IDL. The name attribute is required, except when the structure property is used in a sequence.

type

The **type** attribute specifies the type of the corresponding IDL struct.

69.8.2.10 The value Element

The **value** element is used to specify a simple value.

<!ELEMENT value (#PCDATA) >

69.9 Component Deployment

Components, component homes, and component assemblies are deployed on target hosts in a network using a deployment tool provided by an ORB or tool vendor.

The aim of deployment is to install and “hook-up” a logical component topology to a physical computing environment. The deployment is specified by an assembly file, or in the degenerate case, an individual component file.

The basic steps in the deployment process are:

1. Identify on which hosts the components are to be installed. This information will most likely come from an interaction between tool and user. Components are deployed either singly or together with other components as part of a process or host collocation.
2. Install component implementations on each platform where corresponding component instances are to be deployed. If a component implementation, uniquely identified by a UUID, is already installed on a host then it does not have to be installed again.
3. Instantiate components and component homes on particular hosts. The mapping for doing so was determined in step 1.
4. Connect components as specified in the assembly descriptor’s connect block.

A stand-alone component file may be deployed as well as assembly files. In that case, step 4 does not apply. Unless otherwise noted, all interfaces defined in the subsequent sections are in the **Deployment** module which is imbedded within the **Components** module (see Section 695.1, “softpkg.dtd,” on page 695-335 for a description of the naming structure proposed by this specification).

69.9.1 Participants in Deployment

The deployment of a component or component assembly is carried out by a deployment application in conjunction with a set of helper objects. The helper objects include component repositories, assembly and component factories, an object representing an assembly itself, and a container.

The following class diagram and scenario represents a deployment architecture.

Note – Of the interfaces described below, only **ComponentInstallation**, **AssemblyFactory**, and **Assembly** are required by this specification; the other interfaces are included for illustrative purposes and to support an end-to-end scenario.

69.9.1.1 Deployment Architecture

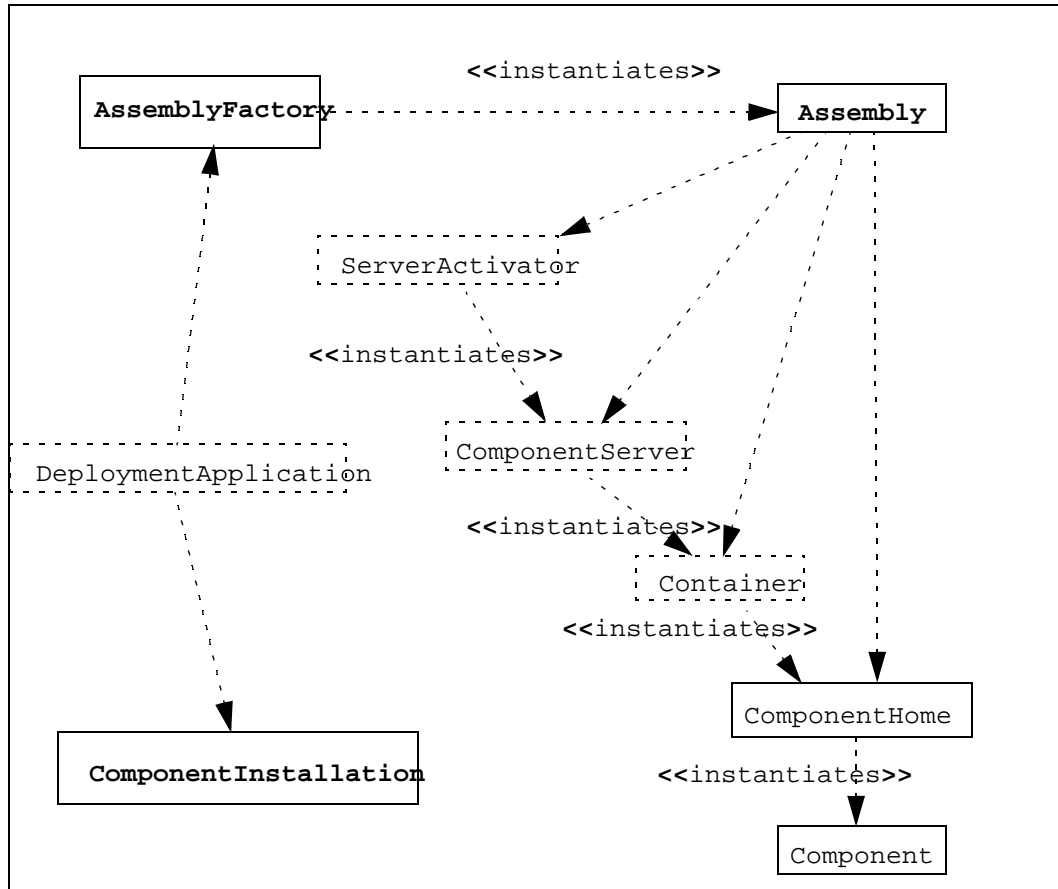


Figure 69-1 Deployment Architecture

69.9.1.2 Deployment Scenario

The steps in deploying and activating a component assembly could unfold as follows.

1. The deployment application has a conversation with the user to determine where each component or collocation is to be placed. Information about where components are to be located is recorded in a copy of the component assembly descriptor. This marked-up assembly descriptor will be used later by the **Assembly** object to direct the creation of the assembly.
2. Next the component implementations are installed on the platforms where they are to be used. The deployment application calls *install* on the **Installation** object, passing the component implementation **id** and a string denoting the address of the component file. If the component has not already been installed on the target platform, then the **Installation** object retrieves the component file and makes it available in the local environment.

3. The deployment application then creates an **Assembly** object. **Assembly** objects coordinate the creation and destruction of component assemblies. Each **Assembly** object represents an assembly instantiation. **Assembly** objects are created by calling an **AssemblyFactory** object on the host where the assembly object is to be created. The **AssemblyFactory** is passed a string pointing to the assembly descriptor file. If necessary, the **AssemblyFactory** brings the assembly descriptor into the local environment and makes its location known to the **Assembly** object.
4. The assembly descriptor uses the assembly descriptor as a recipe for creating the assembly. The descriptor specifies which components and component homes to create, where they are to be located, what components are to be collocated with each other, and what components are to be connected with each other. Based on this information the **Assembly** object creates each component and component home and “hooks-up” the assembly.
5. In creating a component, the **Assembly** object must create a component server, create a container within the server, install a home object within the container, and then use the home to create the component. This work is completed with the help of a set of objects on each host. These are **ServerActivator**, **ComponentServer**, **Container**, and the **ComponentHome**.
6. The **Assembly** object first calls the **ServerActivator** on the target host to create the component server. There is one instance of the **ServerActivator** object on each host. The **Assembly** object creates the component server by calling the **create_component_server** operation on the **ServerActivator** object. This operation creates an empty server process and returns a reference to the **ComponentServer** object of the newly created process.
7. Each server contains a single **ComponentServer** object. It is used by the **Assembly** object to create containers within the server. A container is created when the **Assembly** object calls **create_container** on the **ComponentServer** object, passing in a container identifier or list of container attributes. The **create_container** operation returns a reference to the **Container** interface of the newly created container.
8. The **Assembly** object uses the **Container** interface to install the component home into the container. This is accomplished by calling **install_home** on the **Container** object. The **install_home** operation takes a component **id** parameter and returns a reference to the home interface.
9. In order to create the home, the **Container** must load the DLL, shared object file, or .class file into the container process. To determine the path or the fully qualified name of the component implementation, the container calls the **get_implementation** operation of the **Installation** object. It passes in the **id** of the component implementation and is returned the absolute location or name of the component implementation. The container then loads the implementation and instantiates the home object. The home object reference is then returned to the **Assembly** object.

10. The **Assembly** object uses the component's home object to create the component instance. The instance is created by calling **create_component** on the home reference. **create_component** returns a **CCMObject** object reference.
11. If applicable, a configurator is applied to the component.
12. Once all of the components are installed, the **Assembly** object connects components in the assembly based on the information in the connect block of the assembly descriptor. It does this by calling the receptacle connect operation on the **CCMObject** reference.
13. Following the successful consummation of each connection in the assembly, the **Assembly** object calls **configuration_complete** on each object in the assembly to signal that all of its initial connections have been fixed.

69.9.2 ComponentInstallation Interface

The **ComponentInstallation** object is used to install, query, and remove component implementations on a single platform. There is at most one **ComponentInstallation** object per host.

It is intended that this interface be general enough to encompass a wide range of underlying implementations, as the **ComponentInstallation** interface will likely be implemented on top of a vendor specific implementation repository.

```

exception UnknownImplId { };
exception InvalidLocation { };

interface ComponentInstallation {
    boolean install(in string implUUID, in string component_loc)
        raises InvalidLocation;
    boolean replace(in string implUUID, in string component_loc)
        raises InvalidLocation;
    boolean remove(in string implUUID)
        raises UnknownImplId;
};

```

install

The **install** operation installs a component on the particular host on which the **ComponentInstallation** object resides. The **component_loc** parameter points to the location of the component package. The **implUUID** refers to a particular implementation within that component package.

replace

The **replace** operation replaces a component implementation previously installed. The **component_loc** parameter points to the component package and the **implUUID** points to a particular implementation within the package.

remove

The **remove** operation removes a previously installed component implementation. The **implUUID** refers to the particular implementation.

69.9.3 *AssemblyFactory Interface*

The **AssemblyFactory** interface is used to create **Assembly** objects. A single **AssemblyFactory** object must be present on each host where **Assembly** objects are to be created.

```

                exception InvalidLocation { };
exception InvalidAssembly { };

interface AssemblyFactory {
    Cookie create(in string assembly_loc)
        raises InvalidLocation;
    Assembly lookup(in Cookie c)
        raises InvalidAssembly;
    boolean destroy(in Cookie c)
        raises InvalidAssembly;
};

```

create

The **create** operation creates an **Assembly** object on the host on which the **AssemblyFactory** is located. It takes a string location for the assembly descriptor and returns a **Cookie** that may be used to reference the assembly later. The **Cookie** is the same as specified in Section 61.5.2.4, “Cookie type,” on page 61-42, of this document. The operation raises an **InvalidLocation** exception if the assembly descriptor could not be found.

lookup

The **lookup** operation takes a **Cookie** and returns an object reference to an **Assembly** object. It throws **InvalidAssembly** exception if the **Cookie** did not reference an existing assembly, known by this **AssemblyFactory**.

destroy

The **destroy** operation destroys the assembly referenced by a **Cookie**. If the assembly is active it will first tear down the assembly. The operation returns true if the assembly was successfully destroyed, false otherwise.

69.9.4 Assembly Interface

The **Assembly** interface represents an assembly instantiation. It is used to build up and tear down component assemblies. Building the assembly means that it is going to instantiate all of the components in the assembly and create connections between them as specified in the assembly descriptor. Tearing the assembly down means removing all connections and destroying the components in the assembly.

```
enum AssemblyState {INACTIVE, INSERVICE};
```

```
interface Assembly {
    boolean build();
    boolean tear_down();
    AssemblyState get_state();
};
```

build

The **build** operation builds the assembly and returns TRUE if the assembly was built successfully and FALSE otherwise. If the build failed then the **build** operation is responsible for cleaning up any pieces of the assembly that were created.

tear_down

The **tear_down** operation removes all of the objects in the assembly. It cannot be responsible for any objects which the assembly objects created during operation.

get_state

The **get_state** operation returns whether the assembly is active or inactive. An assembly will be inactive before it is built, while it is being built, when it is being torn down, and after it has been torn down. It will be active after it is successfully built and before it is torn down.

69.9.5 Component Entry Points (Component Home Factories)

Each component package contains a component implementation. A component implementation is a dynamically loadable module such as a DLL, a shared library, or a Java .class file. The component implementation file contains the code for the component implementation and its associated home implementation.

To load a component into a container, the home for the component must first be created. The home is then used to create component instances. The component's home is created by calling a well known entry point in the component implementation file.

The entry point is an operation or function whose existence and signature is common across all component implementation files. The generic entry point function allows a container to create a component home without having to have specific knowledge of that home or its associated component implementation.

Entry points are programming language specific. Depending on the language, it is either a function or static method. The signature and semantics of the operation are specified for Java and C++.

In general, the entry point function takes no arguments and returns a pointer or reference to a **HomeExecutorBase**.

Entry Points in Java

In Java, the entry point is the name of a class and static method which may be invoked to create a servant which implements the component home. The method must have the following signature:

```
public static HomeExecutorBase
foo();
```

For instance, if one wrote the following code for the entry point:

```
package bigbank.corbacomponents.Account;
public class AccountHomeFactory {
    public static HomeExecutorBase create() {
        return new AccountHomeImpl();
    }
}
```

Then the string representing the entry point string would be “bigbank.corbacomponents.Account.AccountHomeFactory.create”.

Entry Points in C++

In C++, the entry point is the symbol in a shared library or DLL which should be invoked to return the **HomeExecutorBase** for the component’s home implementation.

The entry point should have “C” linkage (i.e. no name-mangling) and have the following signature:

```
HomeExecutorBase* (*)();
```

So for example:

```
extern "C" {
    HomeExecutorBase* createAccountHome() {
        return new AccountHomeImpl();
    };
};
```

In this case, the entry point would simply be “createAccountHome”.

This chapter contains the definitions of the XML DTDs used by the CORBA Components.

Issue – It contains all new text taken from the CCM final submission orbos/99-07-01 with only minor editorial changes - Jeff Mischkinsky

This chapter contains the following sections.

| Section Title | Page |
|-------------------------|---------|
| “softpkg.dtd” | 695-335 |
| “corbacomponent.dtd” | 695-339 |
| “properties.dtd” | 695-345 |
| “componentassembly.dtd” | 695-346 |
| | |

695.1 softpkg.dtd

```
<!-- DTD for softpkg. Used to describe CORBA Component
      implementations. The root element is <softpkg>.
      Elements are listed alphabetically.
-->
<!-- Simple xml link attributes based on W3C WD-xlink-19980303.
      May change when XML is finalized. -->
<!ENTITY % simple-link-attributes "
```

```

        xml:link      CDATA      #FIXED 'SIMPLE'
        href          CDATA      #REQUIRED
    ">

<!--ELEMENT author
    ( name
      | company
      | webpage
    ) * >

<!--ELEMENT code
    ( ( codebase
      | fileinarchive
      | link
      )
      , entrypoint?
      , usage?
    ) >
<!--ATTLIST code
    type CDATA #IMPLIED >

<!-- If file not available locally, then download via codebase link -->
<!--ELEMENT codebase EMPTY >
<!--ATTLIST codebase
    filename CDATA #IMPLIED
    %simple-link-attributes; >

<!--ELEMENT compiler EMPTY >
<!--ATTLIST compiler
    name      CDATA #REQUIRED
    version   CDATA #IMPLIED >

<!--ELEMENT company ( #PCDATA ) >

<!--ELEMENT dependency
    ( softpkgref
      | codebase
      | fileinarchive
      | localfile
      | name
      ) >
<!--ATTLIST dependency
    type      CDATA      #IMPLIED
    action (assert | install) "assert">

<!--ELEMENT description ( #PCDATA ) >

<!--ELEMENT descriptor
    ( link
      | fileinarchive
      ) >
<!--ATTLIST descriptor
    type      CDATA #IMPLIED>

<!--ELEMENT entrypoint ( #PCDATA ) >

```



```

<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
    class      CDATA      #REQUIRED
    origin     CDATA      #REQUIRED
    id         ID         #IMPLIED
    extra      CDATA      #IMPLIED
    html-form  CDATA      #IMPLIED >

<!-- The "fileinarchive" element is used to specify a file in the
archive.
    If the file is in another archive then link
    is used to point to the archive in which the file may be found.
-->
<!ELEMENT fileinarchive
    ( link? ) >
<!ATTLIST fileinarchive
    name CDATA #REQUIRED >

<!ELEMENT idl
    ( link
    | fileinarchive
    | repository
    ) >

<!ELEMENT implementation
    ( description
    | code
    | compiler
    | dependency
    | descriptor
    | extension
    | programminglanguage
    | humanlanguage
    | os
    | propertyfile
    | processor
    | runtime
    )* >
<!ATTLIST implementation
    id ID #IMPLIED >

<!ELEMENT implref EMPTY >
<!ATTLIST implref
    idref CDATA #REQUIRED >

<!ELEMENT humanlanguage EMPTY >
<!ATTLIST humanlanguage
    name CDATA #REQUIRED >

<!ELEMENT license ( #PCDATA ) >
<!ATTLIST license
    %simple-link-attributes; >

```

```
<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
    %simple-link-attributes; >

<!-- A file that should be available in the local environment -->
<!ELEMENT localfile EMPTY >
<!ATTLIST localfile
    name CDATA #REQUIRED >

<!ELEMENT name ( #PCDATA ) >

<!ELEMENT os EMPTY >
<!ATTLIST os
    name CDATA #REQUIRED
    version CDATA #IMPLIED>

<!ELEMENT pkgtype ( #PCDATA ) >
<!ATTLIST pkgtype
    version CDATA #IMPLIED >

<!ELEMENT processor EMPTY >
<!ATTLIST processor
    name CDATA #REQUIRED >

<!ELEMENT programminglanguage EMPTY>
<!ATTLIST programminglanguage
    name CDATA #REQUIRED
    version CDATA #IMPLIED >

<!ELEMENT propertyfile
    ( fileinarchive
      | link) >
<!ATTLIST propertyfile
    type CDATA #IMPLIED >

<!ELEMENT resource
    ( localfile
      | codebase
    ) >

<!ATTLIST resource
    type CDATA #IMPLIED >

<!ELEMENT runtime EMPTY >
<!ATTLIST runtime
    name CDATA #REQUIRED
    version CDATA #IMPLIED>

<!ELEMENT softpkg
    ( title
      | pkgtype
      | author
      | description?
      | license
      | idl
```

```

        | propertyfile
        | dependency
        | descriptor
        | implementation
        | extension
    ) * >
<!ATTLIST softpkg
    name      ID      #REQUIRED
    version   CDATA   #IMPLIED >

<!ELEMENT softpkgref
    ( ( fileinarchive
        | link
        )
      , implref?
    ) >

<!ELEMENT title ( #PCDATA ) >

<!ELEMENT usage ( #PCDATA ) >

<!ELEMENT webpage ( #PCDATA ) >
<!ATTLIST webpage
    %simple-link-attributes; >

```

695.2 corbacomponent.dtd

```

<!-- DTD for CORBA Component Descriptor. The root element is
      <corbacomponent>. Elements are listed alphabetically.
-->

<!-- Simple xml link attributes based on W3C WD-xlink-19980303.
      May change when XML is finalized. -->
<ENTITY % simple-link-attributes "
    xml:link      CDATA          #FIXED 'SIMPLE'
    href          CDATA          #REQUIRED
">

<!ELEMENT accessmode EMPTY>
<!ATTLIST accessmode
    mode (READ_ONLY|READ_WRITE) #REQUIRED >

<!ELEMENT catalog EMPTY>
<!ATTLIST catalog
    type CDATA #REQUIRED >

<!ELEMENT componentfeatures
    ( inheritscomponent?
      , supportsinterface*
      , ports
      , operationpolicies?
      , extension*
    ) >

```

```

    ) >
<!ATTLIST componentfeatures
    name CDATA #REQUIRED
    repid CDATA #REQUIRED >

<!ELEMENT componentkind
    ( service
    | session
    | process
    | entity
    | unclassified
    ) >

<!ELEMENT componentproperties
    ( fileinarchive
    ) >

<!ELEMENT componentrepid EMPTY >
<!ATTLIST componentrepid
    repid CDATA #IMPLIED >

<!ELEMENT containermanagedpersistence
    ( storagehome
    , pssimplementation?
    , catalog?
    , accessmode
    , psstransaction
    , params?
    ) >

<!ELEMENT configurationcomplete EMPTY >
<!ATTLIST configurationcomplete
    set ( true | false ) #REQUIRED >

<!ELEMENT consumes
    ( eventpolicy
    , extension* ) >
<!ATTLIST consumes
    consumesname CDATA #REQUIRED
    eventtype CDATA #REQUIRED >

<!ELEMENT corbacomponent
    ( corbaversion
    , componentrepid
    , homerepid
    , componentkind
    , interop?
    , transaction?
    , security?
    , threading
    , configurationcomplete
    , extendedpoapolicy*
    , repository?
    , segment*
    , componentproperties?

```

```

        , homeproperties?
        , homefeatures+
        , componentfeatures+
        , interface*
        , extension*
    ) >

<!ELEMENT corbaversion (#PCDATA) >

<!ELEMENT emits
    ( eventpolicy
      , extension* ) >
<!ATTLIST emits
    emitsname CDATA #REQUIRED
    eventtype CDATA #REQUIRED >

<!ELEMENT entity
    ( servant ) >

<!ELEMENT eventpolicy EMPTY>
<!ATTLIST eventpolicy
    policy ( normal | default | transaction ) #IMPLIED>

<!ELEMENT extendedpoapolicy EMPTY>
<!ATTLIST extendedpoapolicy
    name CDATA #REQUIRED
    value CDATA #REQUIRED >

<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
    class CDATA #REQUIRED
    origin CDATA #REQUIRED
    id ID #IMPLIED
    extra CDATA #IMPLIED
    html-form CDATA #IMPLIED >

<!-- The "fileinarchive" element is used to specify a file in the
archive.
    If the file is in another archive then link
    is used to point to the archive in which the file may be found.
-->
<!ELEMENT fileinarchive
    ( link? ) >
<!ATTLIST fileinarchive
    name CDATA #REQUIRED >

<!ELEMENT homefeatures
    ( inheritshome?
      , operationpolicies?
      , extension* ) >
<!ATTLIST homefeatures
    name CDATA #REQUIRED
    repid CDATA #REQUIRED >

```

```

<!ELEMENT homeproperties
  ( fileinarchive
    ) >

<!ELEMENT homerepid EMPTY >
<!ATTLIST homerepid
  repid CDATA #IMPLIED >

<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
  repid CDATA #REQUIRED>

<!ELEMENT inheritshome EMPTY>
<!ATTLIST inheritshome
  repid CDATA #REQUIRED>

<!ELEMENT inheritsinterface EMPTY>
<!ATTLIST inheritsinterface
  repid CDATA #REQUIRED>

<!ELEMENT ins EMPTY>
<!ATTLIST ins
  name CDATA #REQUIRED >

<!ELEMENT interface
  ( inheritsinterface*
    , operationpolicies? ) >
<!ATTLIST interface
  name CDATA #REQUIRED
  repid CDATA #REQUIRED >

<!ELEMENT interop EMPTY>
<!ATTLIST interop
  type CDATA #REQUIRED
  direction ( hasview | isview ) #REQUIRED
  descriptor CDATA #REQUIRED >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
  %simple-link-attributes; >

<!ELEMENT objref EMPTY>
<!ATTLIST objref
  string CDATA #REQUIRED >

<!ELEMENT operation
  ( transaction?
    , requiredrights? ) >
<!ATTLIST operation
  name CDATA #REQUIRED >
<!-- an operation name of "*" specifies all operations in the current
scope -->

<!ELEMENT operationpolicies
  ( operation+ ) >

```

```

<!ELEMENT param EMPTY >
<!ATTLIST param
    name CDATA #REQUIRED
    value CDATA #REQUIRED >

<!ELEMENT params (param+) >

<!ELEMENT poapolicies EMPTY>
<!ATTLIST poapolicies
    thread (ORB_CTRL_MODEL | SINGLE_THREAD_SAFE ) #REQUIRED
    lifespan (TRANSIENT | PERSISTENT ) #REQUIRED
    iduniqueness (UNIQUE_ID | MULTIPLE_ID) #REQUIRED
    idassignment (USER_ID | SYSTEM_ID) #REQUIRED
    servantretention (RETAIN | NON_RETAIN) #REQUIRED
    requestprocessing (USE_ACTIVE_OBJECT_MAP_ONLY
        | USE_DEFAULT_SERVANT
        | USE_SERVANT_MANAGER) #REQUIRED
    implicitactivation (IMPLICIT_ACTIVATION
        | NON_IMPLICIT_ACTIVATION) #REQUIRED >

<!ELEMENT ports
    ( uses
    | provides
    | emits
    | publishes
    | consumes
    )* >

<!ELEMENT process
    ( servant ) >

<!ELEMENT provides
    ( operationpolicies?
    , extension* ) >
<!ATTLIST provides
    providesname CDATA #REQUIRED
    repid CDATA #REQUIRED
    facettag CDATA #REQUIRED >

<!ELEMENT pssimplementation EMPTY>
<!ATTLIST pssimplementation
    id CDATA #REQUIRED >

<!ELEMENT psstransaction (psstransactionisolationlevel?) >
<!ATTLIST psstransaction
    policy (TRANSACTIONAL|NON_TRANSACTIONAL) #REQUIRED >

<!ELEMENT psstransactionisolationlevel EMPTY>
<!ATTLIST psstransactionisolationlevel
    level (READ_UNCOMMITTED|READ_COMMITTED|REPEATABLE_READ|SERIALIZABLE)
    #REQUIRED >

<!ELEMENT publishes
    ( eventpolicy

```

```

        , extension* ) >
<!ATTLIST publishes
    publishesname CDATA #REQUIRED
    eventtype CDATA #REQUIRED >

<!ELEMENT repository
    ( ins
      | objref
      | link
    ) >
<!ATTLIST repository
    type CDATA #IMPLIED >

<!ELEMENT requiredrights
    ( right* ) >

<!ELEMENT right
    ( description? ) >
<!ATTLIST right
    name CDATA #REQUIRED >

<!ELEMENT security
    ( requiredrights? ) >
<!ATTLIST security
    rightsfamily CDATA #REQUIRED >

<!ELEMENT segment
    ( segmentmember+
      , containermanagedpersistence?
      , extension*
    ) >
<!ATTLIST segment
    name CDATA #REQUIRED
    segmenttag CDATA #REQUIRED >

<!ELEMENT segmentmember EMPTY>
<!ATTLIST segmentmember
    facettag CDATA #REQUIRED >

<!ELEMENT servant EMPTY >
<!ATTLIST servant
    lifetime ( component|method|transaction|container ) #REQUIRED >

<!ELEMENT service EMPTY >

<!ELEMENT session
    ( servant ) >

<!ELEMENT storagehome EMPTY>
<!ATTLIST storagehome
    id CDATA #REQUIRED >

<!ELEMENT supportsinterface
    ( operationpolicies?
      , extension* ) >

```



```

<!ATTLIST supportsinterface
    repid CDATA #REQUIRED >

<!ELEMENT threading EMPTY>
<!ATTLIST threading
    policy ( serialize | multithread ) #REQUIRED >

<!ELEMENT transaction EMPTY >
<!ATTLIST transaction
    use (self-managed|not-supported|required|supports|requires-
new|mandatory|never) #REQUIRED >

<!ELEMENT unclassified
    ( poapolicies ) >

<!ELEMENT uses ( extension* ) >
<!ATTLIST uses
    usesname CDATA #REQUIRED
    repid    CDATA #REQUIRED >

```

695.3 *properties.dtd*

```

<!-- DTD for CORBA Component property file. The root element
    is <properties>. Elements are listed alphabetically.
-->

<!ELEMENT choice ( #PCDATA ) >

<!ELEMENT choices ( choice+ ) >

<!ELEMENT defaultvalue ( #PCDATA ) >

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT value ( #PCDATA ) >

<!ELEMENT properties
    ( description?
      , ( simple
        | sequence
        | struct
        ) *
    ) >

<!ELEMENT simple
    ( description?
      , value
      , choices?
      , defaultvalue?
    ) >

```

```
<!ATTLIST simple
  name CDATA #IMPLIED
  type ( boolean
        | char
        | double
        | float
        | short
        | long
        | objref
        | octet
        | short
        | string
        | ulong
        | ushort
        ) #REQUIRED >

<!ELEMENT sequence
  ( description?
    , ( simple*
      | struct*
      | sequence*
      )
    )
  >

<!ATTLIST sequence
  name CDATA #IMPLIED
  type CDATA #REQUIRED >

<!ELEMENT struct
  ( description?
    , ( simple
      | sequence
      | struct
      )*
    )
  >

<!ATTLIST struct
  name CDATA #IMPLIED
  type CDATA #REQUIRED >
```

695.4 *componentassembly.dtd*

```
<!-- DTD for Component Assembly Descriptor. The root element
      is <componentassembly>. Elements are listed
      alphabetically.
-->

<!-- Simple xml link attributes based on W3C WD-xlink-19980303.
      May change slightly when XLL is finalized.
-->
<!ENTITY % simple-link-attributes "
      xml:link          CDATA          #FIXED 'SIMPLE'
```

```

        href          CDATA          #REQUIRED " " >

<!-- If file not available locally, then download via codebase link -->
<!ELEMENT codebase EMPTY >
<!ATTLIST codebase
    filename CDATA #IMPLIED
    %simple-link-attributes; >

<!ELEMENT componentassembly
    ( description?
      , componentfiles
      , partitioning
      , connections?
      , extension*
    ) >
<!ATTLIST componentassembly
    id ID #REQUIRED
    derivedfrom CDATA #IMPLIED >

<!ELEMENT componentfile
    ( fileinarchive
      | codebase
      | link
    ) >
<!ATTLIST componentfile
    id ID #REQUIRED
    type CDATA #IMPLIED >

<!ELEMENT componentfileref EMPTY >
<!ATTLIST componentfileref
    idref CDATA #REQUIRED >

<!ELEMENT componentfiles
    ( componentfile+
    ) >

<!ELEMENT componentimplref EMPTY >
<!ATTLIST componentimplref
    idref CDATA #REQUIRED >

<!ELEMENT componentinstantiation
    ( usagename?
      , componentproperties?
      , registercomponent*
      , extension*
    ) >
<!ATTLIST componentinstantiation
    id ID #REQUIRED >

<!ELEMENT componentinstantiationref EMPTY >
<!ATTLIST componentinstantiationref
    idref CDATA #REQUIRED >

<!ELEMENT componentproperties
    ( fileinarchive

```

```
        | codebase
      ) >

<!ELEMENT componentsupportedinterface
  ( componentinstantiationref
    | findby
  )>

<!ELEMENT connectevent
  ( consumesport
    , ( emitsport
      | publishesport
    )
  ) >
<!ATTLIST connectevent
  id ID #IMPLIED >

<!ELEMENT connecthomes
  ( proxyhome
    , destinationhome
  ) >
<!ATTLIST connecthomes
  id ID #IMPLIED >

<!ELEMENT connectinterface
  ( usesport
    , ( providesport
      | componentsupportedinterface
      | existinginterface
      | homeinterface
    )
  ) >
<!ATTLIST connectinterface
  id ID #IMPLIED >

<!ELEMENT connections
  ( connectinterface
    | connectevent
    | connecthome
    | extension
  )* >

<!ELEMENT consumesidentifier ( #PCDATA ) >

<!ELEMENT consumesport
  ( consumesidentifier
    , ( componentinstantiationref
      | findby
    )
  )>

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT destination ( #PCDATA ) >
```

```

<!ELEMENT destinationhome
  ( homeplacementref
    | findby
  ) >

<!ELEMENT emitsidentifier ( #PCDATA ) >

<!ELEMENT emitsport
  ( emitsidentifier
    , ( componentinstantiationref
      | findby
    )
  )>

<!ELEMENT executableplacement
  ( usagename?
    , componentfileref
    , componentimplref
    , invocation?
    , destination?
    , extension*
  ) >

<!ATTLIST executableplacement
  id          ID          #REQUIRED
  cardinality CDATA "1" >

<!ELEMENT existinginterface
  ( findby )>

<!-- The "extension" element is used for vendor-specific extensions -->
<!ELEMENT extension (#PCDATA) >
<!ATTLIST extension
  class      CDATA      #REQUIRED
  origin     CDATA      #REQUIRED
  id         ID         #IMPLIED
  extra      CDATA      #IMPLIED
  html-form  CDATA      #IMPLIED >

<!-- The "fileinarchive" element is used to specify a file in the
archive.
      If the file is independent of an archive then link is used to point
to
      the archive in which the file may be found.
-->
<!ELEMENT fileinarchive
  ( link? ) >
<!ATTLIST fileinarchive
  name CDATA #REQUIRED >

<!ELEMENT findby
  ( namingservice
    | stringifiedobjectref
    | traderquery
    | homefinder
  )

```

```

        | extension
      ) >

<!ELEMENT homefinder EMPTY >
<!ATTLIST homefinder
  name CDATA #REQUIRED >

<!ELEMENT homeinterface
  ( homeplacementref
    | findby
  )>

<!ELEMENT homeplacement
  ( usagename?
    , componentfileref
    , componentimplref?
    , homeproperties?
    , componentproperties?
    , registerwithhomefinder*
    , registerwithnaming*
    , registerwithtrader*
    , componentinstantiation*
    , destination?
    , extension*
  ) >
<!ATTLIST homeplacement
  id ID #REQUIRED
  cardinality CDATA "1" >

<!ELEMENT homeplacementref EMPTY >
<!ATTLIST homeplacementref
  idref CDATA #REQUIRED >

<!ELEMENT homeproperties
  ( fileinarchive
    | codebase
  ) >

<!ELEMENT hostcollocation
  ( usagename?
    , impltype?
    , ( homeplacement
      | executableplacement
      | processcollocation
      | extension
    )+
    , destination?
  ) >
<!ATTLIST hostcollocation
  id ID #IMPLIED
  cardinality CDATA "1" >

<!ELEMENT impltype EMPTY >
<!ATTLIST impltype
  language CDATA #REQUIRED

```

```

        version CDATA #IMPLIED >

<!ELEMENT invocation EMPTY >
<!ATTLIST invocation
    args CDATA #REQUIRED >

<!ELEMENT link ( #PCDATA ) >
<!ATTLIST link
    %simple-link-attributes; >

<!ELEMENT namingservice EMPTY >
<!ATTLIST namingservice
    name CDATA #REQUIRED >

<!ELEMENT partitioning
    ( homeplacement
    | executableplacement
    | processcollocation
    | hostcollocation
    | extension
    )* >

<!ELEMENT processcollocation
    ( usagename?
    , impltype?
    , ( homeplacement
    | extension
    )+
    , destination?
    ) >
<!ATTLIST processcollocation
    id ID #IMPLIED
    cardinality CDATA "1" >

<!ELEMENT providesidentifier ( #PCDATA ) >

<!ELEMENT providesport
    ( providesidentifier
    , ( componentinstantiationref
    | findby
    )
    ) >

<!ELEMENT publishesidentifier ( #PCDATA ) >

<!ELEMENT publishesport
    ( publishesidentifier
    , ( componentinstantiationref
    | findby
    )
    ) >

<!ELEMENT registercomponent
    ( ( emitsidentifier
    | providesidentifier

```

```

        | publishesidentifier
      )?
    , ( registerwithnaming
      | registerwithtrader
      )+
    ) >

<!ELEMENT registerwithhomefinder EMPTY >
<!ATTLIST registerwithhomefinder
  name CDATA #REQUIRED >

<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming
  name CDATA #IMPLIED >

<!ELEMENT registerwithtrader
  ( traderproperties ) >
<!ATTLIST registerwithtrader
  tradername CDATA #IMPLIED >
<!-- DEVNOTE: is tradername necessary? -->
<!-- DEVNOTE: Should trader properties be specified in component file?
And in assembly file? -->

<!ELEMENT proxyhome
  ( homeplacementref
    | findby
  ) >

<!ELEMENT stringifiedobjectref ( #PCDATA ) >

<!ELEMENT traderconstraint ( #PCDATA ) >

<!ELEMENT traderexport
  ( traderservicetype
    , traderproperties
  ) >

<!ELEMENT traderpolicy
  ( traderpolicyname
    , traderpolicyvalue
  ) >

<!ELEMENT traderpolicyname ( #PCDATA ) >

<!ELEMENT traderpolicyvalue ( #PCDATA ) >

<!ELEMENT traderpreference ( #PCDATA ) >

<!ELEMENT traderproperties
  ( traderproperty+ ) >

<!ELEMENT traderproperty
  ( traderpropertyname
    , traderpropertyvalue

```



```
    ) >

<!ELEMENT traderpropertyname ( #PCDATA ) >

<!ELEMENT traderpropertyvalue ( #PCDATA ) >

<!ELEMENT traderquery
    ( trader servicetype name
      , trader constraint
      , trader preference?
      , trader policy*
      , trader specifiedprop*
    ) >

<!ELEMENT traderservicetype name ( #PCDATA ) >

<!ELEMENT traderspecifiedprop ( #PCDATA ) >

<!ELEMENT usagename ( #PCDATA ) >

<!ELEMENT usesidentifier ( #PCDATA ) >

<!ELEMENT usesport
    ( uses identifier
      , ( component instantiationref
        | findby
      )
    ) >
```

