SCA
Software Communication Architecture

Docente: Gabriele Di Stefano

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila, Italy
gabriele.distefano@univaq.it

Corso per Thales - Chieti

SCA - Introduction

We will see:

1. Overview
2. Architecture Overview
   - Definitions
   - Core Framework
   - Operating Environment
   - Architectural structure of the SCA
   - Networking Overview
3. Operating Environment
   - Operating System
   - CORBA middleware and services
   - Core Framework
   - CF: Base Application Interfaces
   - CF: Base Device Interfaces
   - CF: Framework Control Interfaces
   - CF: Framework Services Interfaces
4. Domain Profiles & Ossie

These slides are available at: http://gs.ing.univaq.it/SCA.

SCA - Motivations

The Software Defined Radio (SDR) is a technology without any prior constraints or design specification.

SDRs are characterized by a significant software component and provide flexibility on the physical layer.

The utilization of this technology by vendors is only practical if a common SDR architecture is defined and a design model is standardized.

The Software Communications Architecture (SCA) is currently one of the most complete and well-defined architecture available for SDRs.

SCA - Aims

The Software Communication Architecture (SCA v.2.2) is published by the Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS), and SCA v.4.0 by the Joint Tactical Networking Center (JTNC).

This architecture was developed to assist in the development of software defined radio communication systems, capturing the benefits of recent technology advances which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs.

The SCA has been structured to:

1. provide for portability of applications software between different SCA implementations,
2. leverage commercial standards to reduce development cost,
3. reduce software development time through the ability to reuse design modules,
4. build on evolving commercial frameworks and architectures.

SCA - What is not

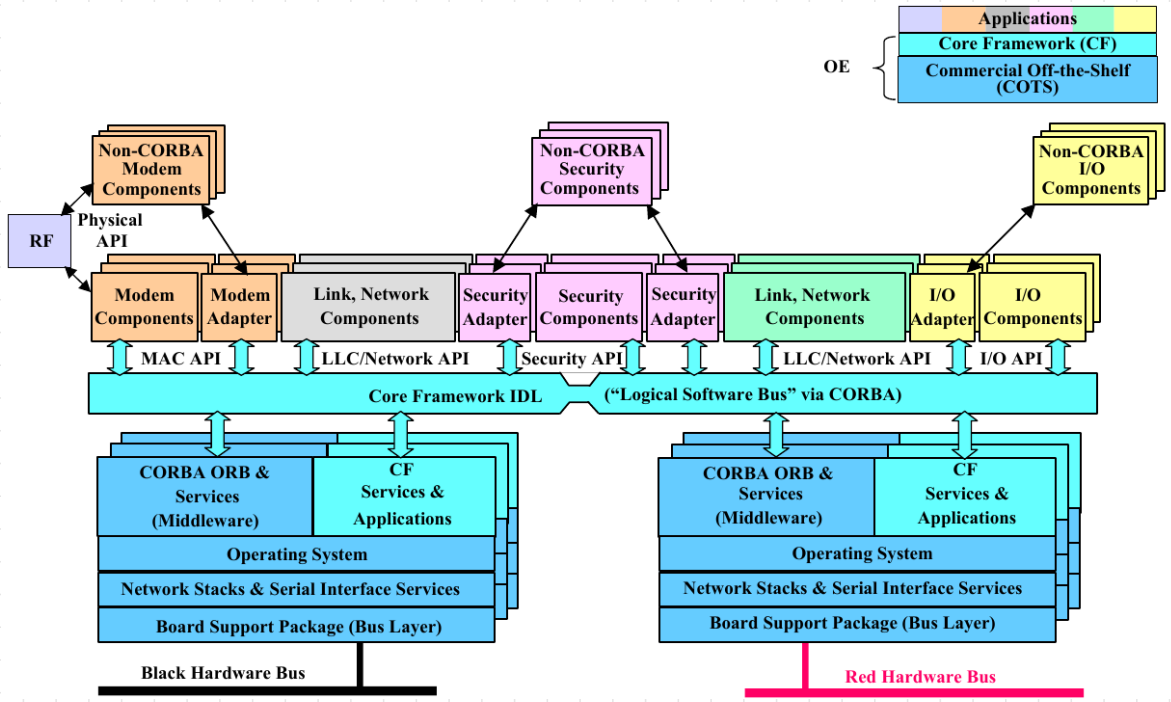
The SCA is not a system specification but an implementation independent set of rules that constrain the design of systems to achieve the objectives listed above.

The SCA establishes an implementation-independent framework with baseline requirements for the development of software for SDRs.

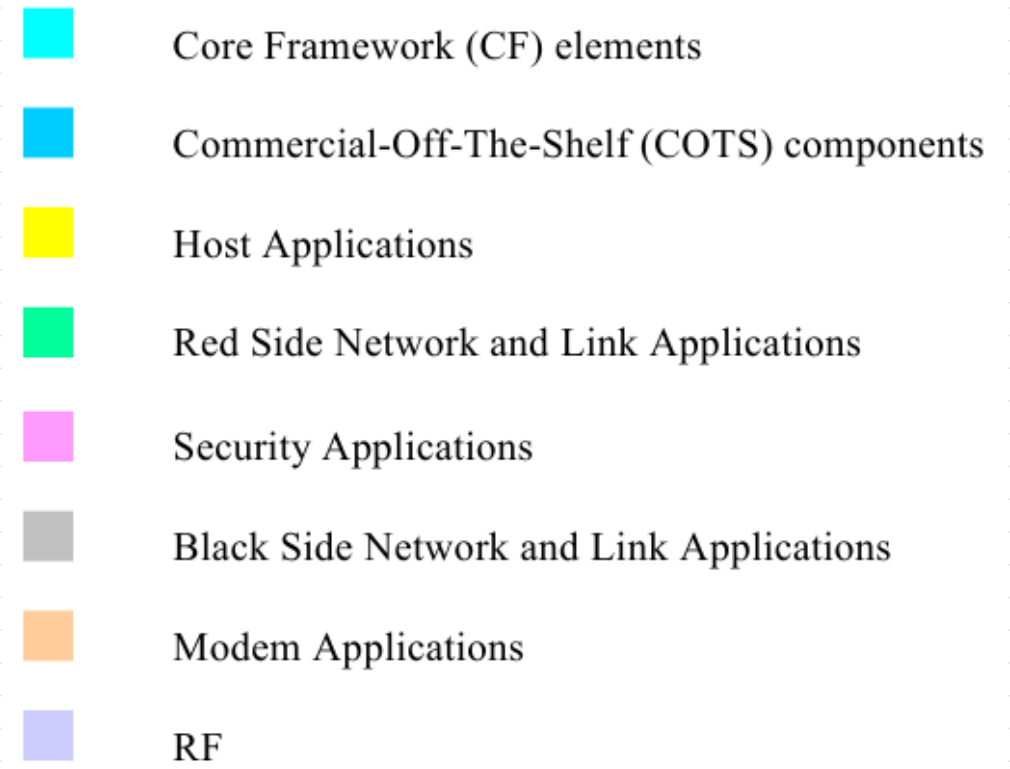
The SCA is an architectural framework that was created to maximize portability, interoperability, and configurability of the software.

Constraints on software development imposed by the framework are on the interfaces and the structure of the software and not on the implementation of the functions.

SCA - Software Structure



SCA - Software Structure (Color Coding)



Role of SCA

Role: to provide a common infrastructure for managing the software and hardware elements present in a system and ensuring that their requirements and capabilities are commensurate.

How: by defining a set of interfaces that isolate the system applications from the underlying hardware.

This set of interfaces is referred to as the Core Framework of the SCA.

Terminology

The word "shall" is used to indicate absolute requirements of this specification which must bstrictly followed in order to achieve compliance. No deviations are permitted.

The phrase "shall not" is used to indicate a strict and absolute prohibition of this specification.

The word "should" is used to indicate a recommended course of action among several possible choices, without mentioning or excluding others. "Should not" is used to discourage a course of action without prohibiting it.

The word "may" is used to indicate a truly optional item or allowable course of action within th scope of the specification. A product which chooses not to implement the indicated item must be able to interoperate with one that does without impairment of required behavior.

Core Framework

The Core Framework is the essential set of open application-layer CORBA interfaces and services which provide an abstraction of the underlying system software and hardware. The Core Framework consists of:

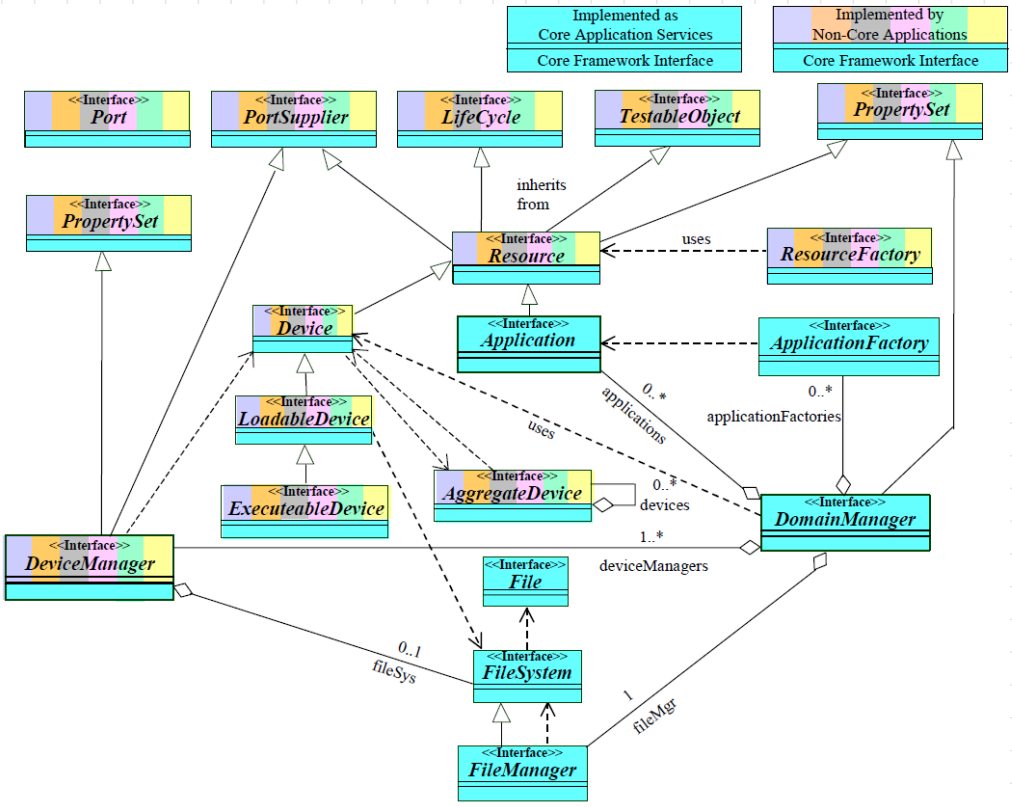
Base Application Interfaces: *Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory, and Resource*, for the management and control interfaces for all system software components.

Base Device Interfaces: *Device, LoadableDevice, ExecutableDevice, and AggregateDevice*, for the management and control of hardware devices within the system through their software interface,

Framework Control Interfaces: *Application, ApplicationFactory, DomainManager, and DeviceManager*, to control the instantiation, management, and destruction/removal of software from the system,

Framework Services Interfaces: *File, FileSystem, and FileManager*, that provide additional support functions and services.

Core Framework IDL Relationships



Operating Environment

The SCA differentiates between the following software components:

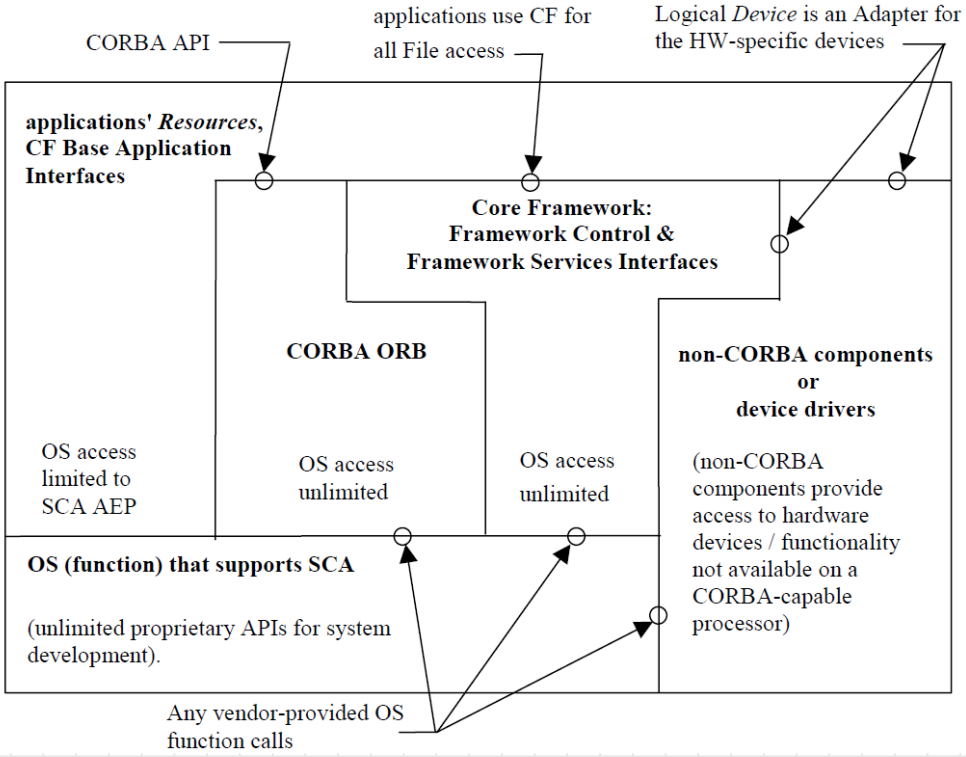
Waveform "application" software: manipulates input data and determines the output of the system. The "application" software implements the Base Application Interfaces.

SCA "devices": provide access to the system hardware resources and implement the Base Device Interfaces.

Services: non-hardware (software-only) resources provided by the system for use by applications.

Operating Environment (OE): software components which provide for the management and execution of the SCA applications and devices. The OE consists of an Operating System (OS), CORBA middleware (including the OMG-defined Event and Naming Services), and the elements defined by the Framework Control and Service Interfaces.

Operating Environment



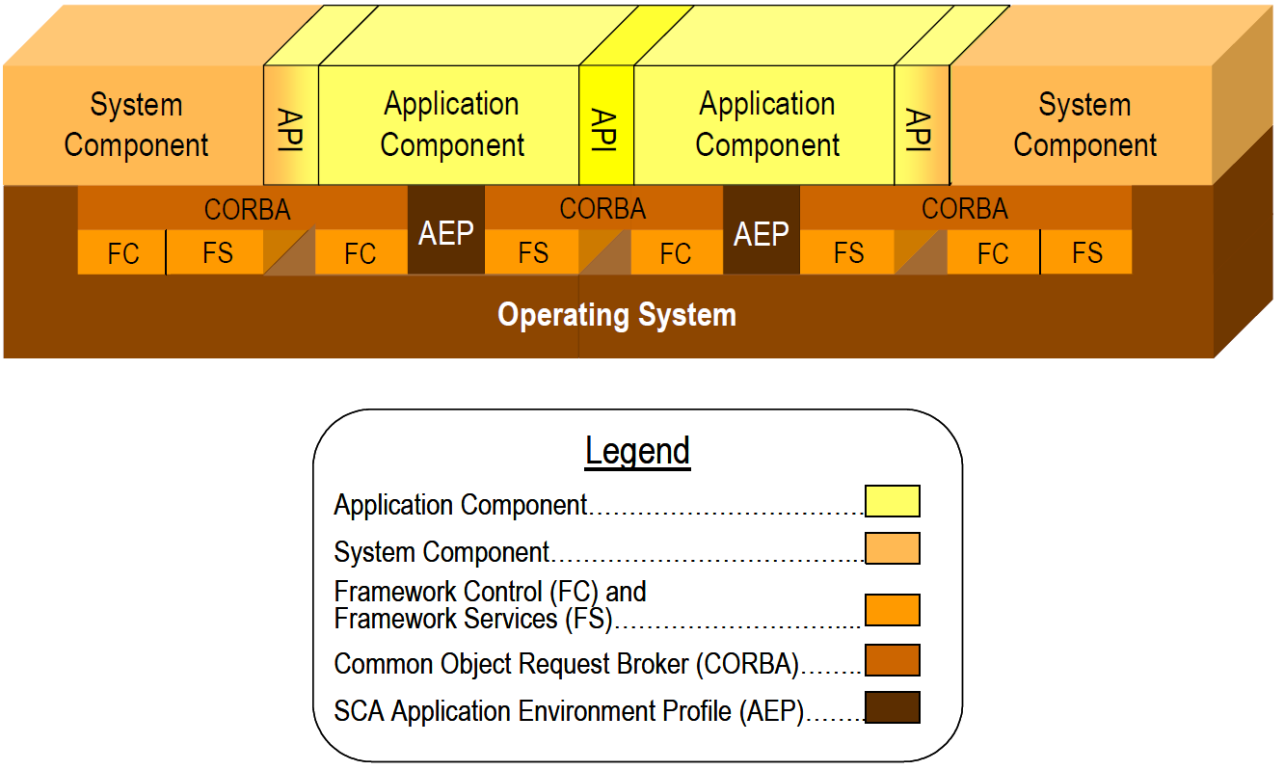
Architectural Structure of the SCA

In the SCA, an application consists of multiple software components that are loaded onto a distributed-processing system.
These components are managed by an implementation of the Framework Control Interfaces.

Application Components communicate either with each other or with the services and devices provided by the system through extensions of the SCA-defined *Port* interface.
Communications between the application and the Framework Services Interfaces are accomplished through the CORBA middleware.
Application may access OS functionality but is restricted to the operations enumerated in the SCA Application Environment Profile (AEP): subset of the Portable Operating System Interface (POSIX) specification.

System Components are limited, managed by the Framework Control Interfaces through the Base Device Interfaces, and are not restricted to functionality of the OS, as are in general system specific.

Architectural Structure of the SCA



Architectural Structure: management hierarchy

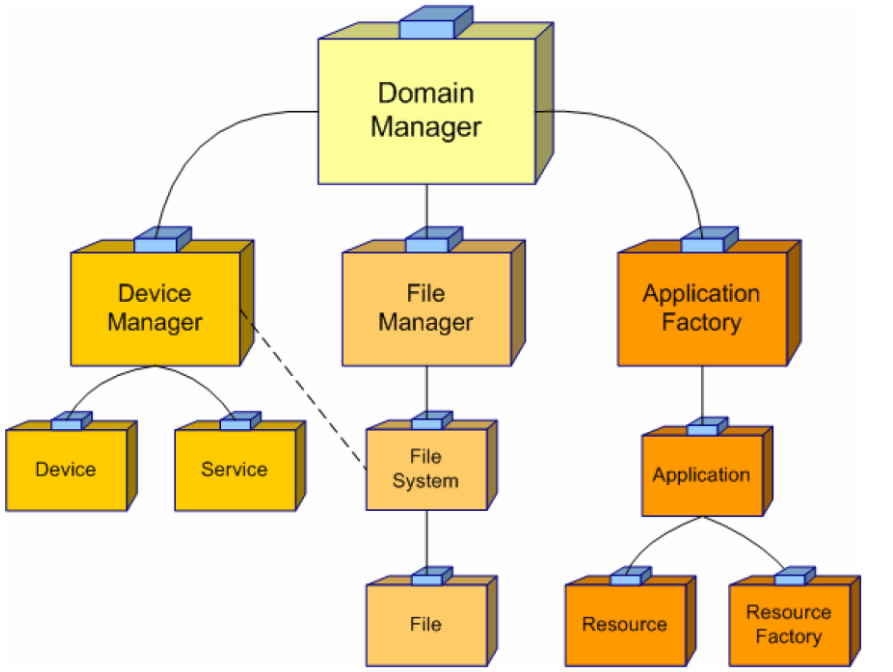
SCA compliant systems require certain software components to be present in order to provide for component deployment, management, and interconnection. These components include the DomainManager, DeviceManager, FileManager, and FileSystem interfaces.

The Domain Manager contains knowledge of all existing implementations installed or loaded onto the system including references to all file systems, device managers, and all applications and their resources.

Each Device Manager, contains complete knowledge of a set of devices and/or services. A system may have multiple Device Managers but each device manager registers with the domain manager.
A Device Manager may have an associated file system.

The implementation of the Application interface (created by the ApplicationFactory) contains all the information regarding a specific application that is instantiated on the system.

SCA Management Hierarchy at Instantiation



Domain Profile

The Domain Profile is a hierarchical collection of eXtensible Markup Language (XML) files that define the properties of all software components (services, devices, applications) in the system.

All CORBA software elements of the system are described by a Software Package Descriptor (SPD) and a Software Component Descriptor (SCD) file.

SPD: provides identification of the software (title, author, etc.) as well as the name of the code file (executable, library or driver), implementation details (language, OS, etc.), configuration and initialization properties (contained in a Properties File), dependencies to other SPDs and devices, and a reference to a Software Component Descriptor.

SCD: defines CORBA interfaces supported and used by a specific component.

Domain Profile: Applications

Since applications are composed of multiple SW components a Software Assembly Descriptor (SAD) file is defined to determine the composition and configuration of the application. The SAD references all SPDs needed for this application, defines required connections between application components (connection of provides and uses ports / interfaces), defines needed connections to devices and services, provides additional information on how to locate the needed devices and services, defines any co-location (deployment) dependencies, and identifies a single component within the application as the assembly controller.

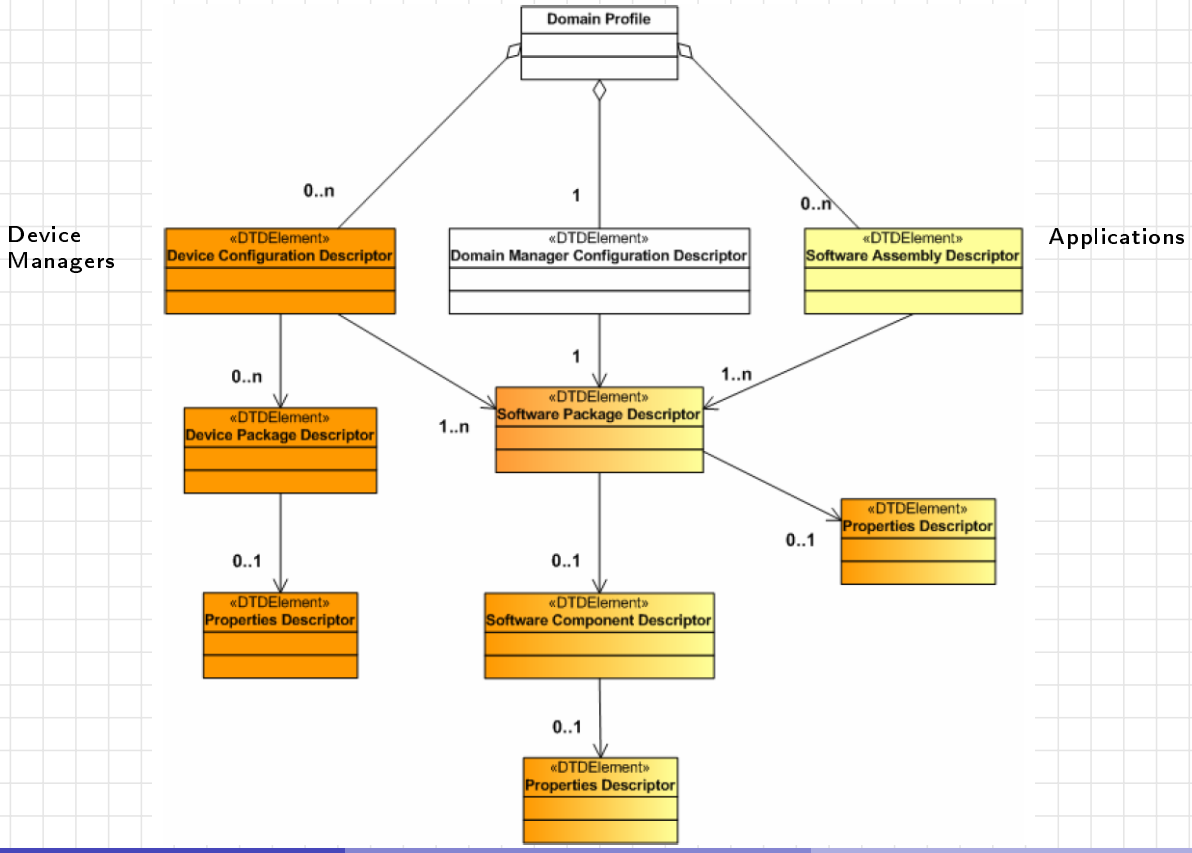
The software profile for an Application consists of one SAD file that references one or more SPD, SCD, and Properties (PRF) files. A PRF file contains information about the properties applicable to a component such as configuration, test, execute, and allocation types.

Domain Profile: Managers

A Device Manager has an associated Device Configuration Descriptor (DCD) file, similar to the application SAD. The DCD identifies all devices and services associated with this device manager, by referencing the associated SPDs. The DCD also defines properties of the specific device manager, enumerates the needed connections to services (e.g. file systems), and provides additional information on how to locate the domain manager. In addition to an SPD, a device may have a Device Package Descriptor (DPD) file which provides a description of the hardware device associated with this (logical) device including description, model, manufacturer, etc.

The implementation of the Domain Manager is itself described by the DomainManager Configuration Descriptor (DMD) which provides the location of the (SPD) file for the specific DomainManager implementation to be loaded. It also specifies the connections to other software components (services and devices) which are required by the domain manager.

Relationship of Domain Profile XML File Types



Operating System

The SCA includes real-time embedded operating system functions, profiled by the Application Environment Profile (AEP) for applications, to provide multi-threaded support for all software executing on the system, including applications, devices, and services.

Appendix B to the SCA Specifications defines the AEP, based on Standardized Application Environment Profile - POSIX Realtime Application Support, IEEE Std 1003.13-2003.

The SCA dictates that an OE provides the options and functions designated as mandatory within the AEP and constrains an application to only use those services. The AEP divides the POSIX options and functions in:

- MAN: the identified function or option is mandatory;
- NRQ: the identified function or option is not required;
- PRT: indicates that only a subset of the indicated option or unit of functionality is required. This designation is followed by a note or cross-reference indicating which elements are required.

Applications

Within the SCA, an Applications consist of one or more resources.
The Resource interface provides a common SCA API for the control and configuration of software components.

Application developers may extend these capabilities by creating specialized Resource interfaces for the application. At a minimum, the extension inherits the Resource interface.

The design of a resource's internal functionality is not dictated by SCA.
This is left to the application developer.

Adapters

Adapters are resources or devices used to support the use of non-CORBA capable elements within the domain.
Adapters are used in an implementation to provide the translation between non-CORBA-capable components or devices and CORBA-capable Resources.

The Adapter concept is based on the industry-accepted Adapter design pattern. Since an Adapter implements the CF CORBA interfaces known to other CORBA-capable Resources, the translation service is transparent to the CORBA-capable Resources.
Adapters become particularly useful to support non-CORBA-capable processing elements.

Reference Model

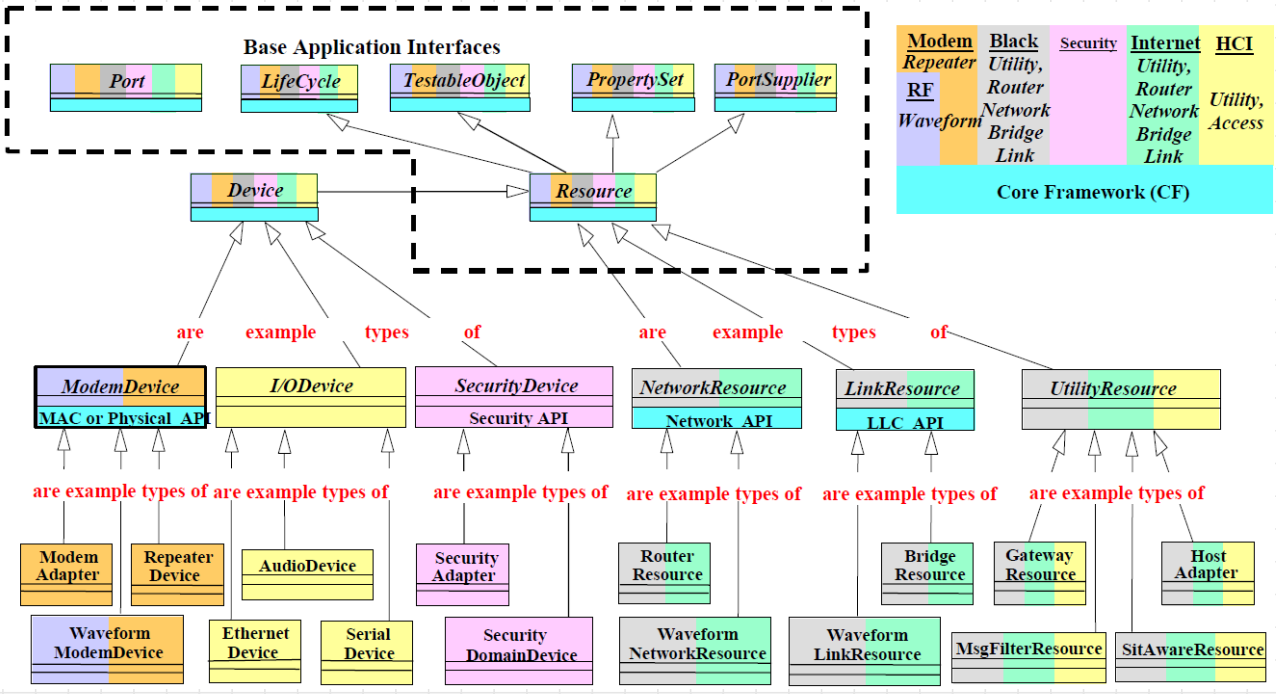
The SCA reference model is realized by defining a standard unit of functionality called a Resource.

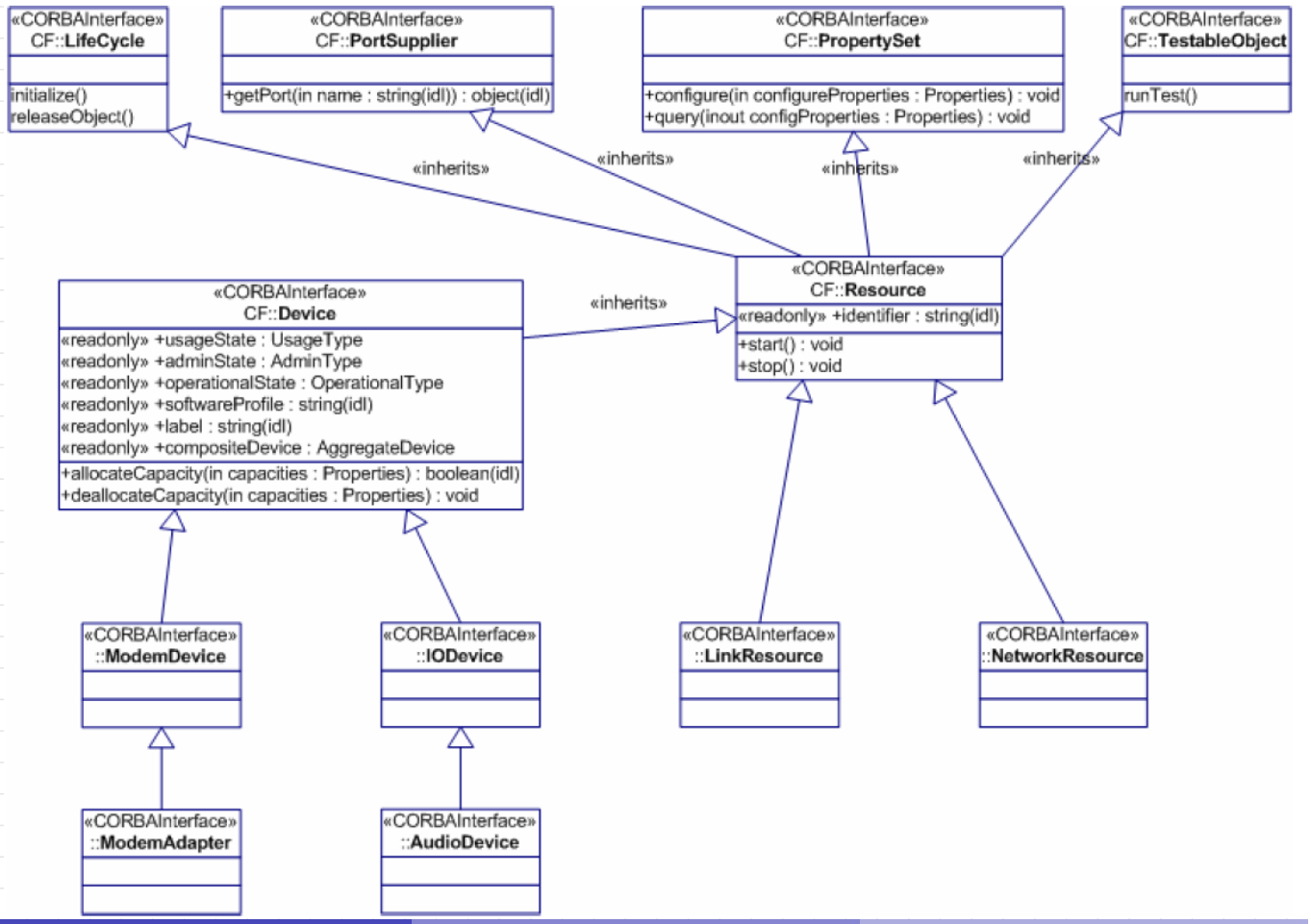
All applications are comprised of resources and using devices.

The operations and attributes provided by the LifeCycle, TestableObject, PortSupplier, and PropertySet interfaces establish a common approach for interacting with any resource in a SCA environment.

The Port interface is used for pushing or pulling messages between resources and devices. A resource may consist of zero or more input and output message ports.

Conceptual model of Resources





Networking Overview

The communications between a SCA-compliant radio system and its peer systems are difined by external networking protocols.

A network of nodes is formed between systems which are interconnected by repeaters, bridges, routers, and/or gateways.

The different categories of interoperability are based upon the OSI Model.

Physical Layer Interoperability: The external networking protocols provide a compatible physical interface, including the signaling interface, but no higher layer processing.

Link Layer Interoperability: The external networking protocols provide link layer processing over all physical interfaces. Intelligent routing or switching decisions are limited.

Network Layer Interoperability: The external networking protocols provide network layer address processing interoperability. The networks being inter-operated are sub-networks of the same Inter-network.

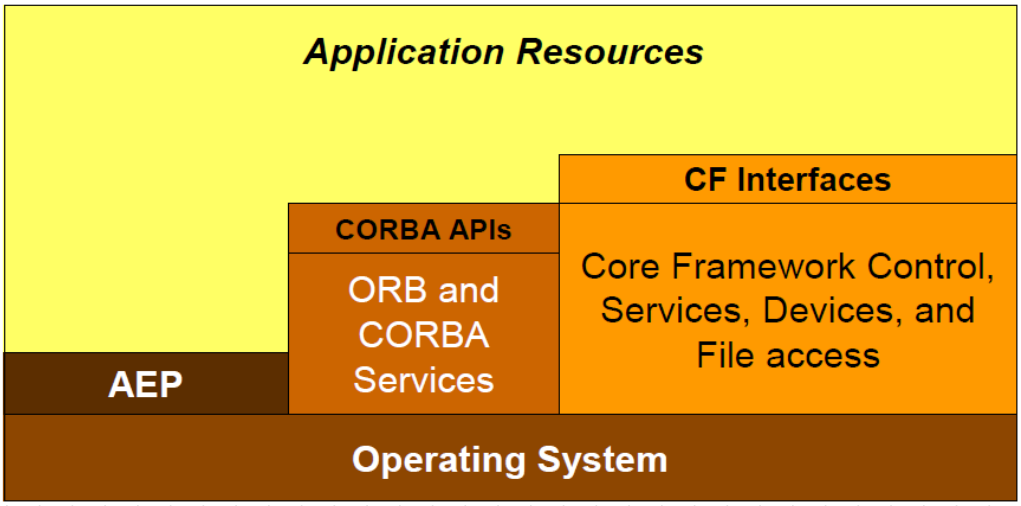
Host Level Interoperability (Layers 4 – 7): Embedded applications can exchange information with hosts attached to the network.

Operating Environment

The Operating Environment consists of:

- the Operating System;
- the CORBA middleware;
- the Core Framework.

Relationships in the Operative Environment



Operating System

The processing environment and the functions performed in the architecture impose differing constraints on the architecture.

An SCA Application Environment Profile (AEP) is defined to support portability of waveforms, scalability of the architecture, and commercial viability.

POSIX specifications are used as a basis for this profile.

The CORBA Object Request Broker (ORB), the CF Framework Control Interfaces, Framework Services Interfaces, and Base Device Interfaces are not limited to using the services designated as mandatory by the SCA AEP.

Applications are limited to using the OS services that are designated as mandatory for AEP. Applications perform file access through the CF.

The OE and related file systems shall support a filename length of 40 characters and a pathname length of 1024 characters.

The Application Environment Profiles (AEP)

The POSIX 1003.13 standard defines four Application Environment Profiles (AEP) PSE-51, PSE-52, PSE-53 and PSE-54.

- PSE-51 – Minimal Real-time System Profile: single process profile with no asynchronous or file Input/Ouput (I/O) specified, typically for embedded controllers.
- PSE-52 – Real-time Controller System Profile: single process profile with asynchronous or file Input/Ouput (I/O), typically for embedded controllers.
- PSE-53 – Dedicated Real-time System Profile: This profile adds multi-process capability to PSE-51.
- PSE-54 – ulti-Purpose Real-time System Profile: This profile includes all the capabilities of the other three profiles and adds multi-user capabilities.

The PSE-SCAS

The PSE-52 AEP was chosen as a starting point, since most operating systems, with guaranteed real time behavior, are of the single process type.

The SCA specifications modifies it and uses it in such a way as to allow multi-process, multi-user operating systems that may conform to PSE-54.

In addition, PSE-52 includes file and file system capability which the SCA needs.

This modified AEP is called PSE-SCAS.

A description of the PSE-SCAS and deviations from PSE-52 are given in MSRC-5000SRD rev. 2.2.

Example of PSE-SCA Specification

Function	AEP
abort()	MAN
alarm()	NRQ
kill()	MAN
pause()	MAN
raise()	MAN
sigaction()	MAN
sigaddset()	MAN
sigdelset()	MAN
sigemptyset()	MAN
sigfillset()	MAN
sigismember()	MAN
signal()	MAN
sigpending()	MAN
sigprocmask()	MAN
sigsupend()	MAN
sigwait()	MAN

MAN: mandatory

NRQ: not required

Remarks on the AEP

- SCA defines a minimum set of POSIX functionality that the OS must provide in order to preserve waveform portability
- SCA does not prohibit the usage of operating systems implementing additional features as they may provide needed or desired functionality for a given domain.
- Additional functionality provided by the OS, must either be abstracted by the Core Framework or be transparent.

CORBA middleware

The OE shall include middleware that, at a minimum, provides the services and capabilities of minimumCORBA as specified by the OMG *Document formal/02-08-01*.

CORBA is used in the CF as the message passing technique for the distributed processing environment.

All CF interfaces are defined in Interface Definition Language (IDL)

The CORBA protocol provides message marshalling to handle the bit packing and handshaking required for delivering, whereas SCA IDL defines operations and attributes that serve as a contract between components.

OE services

The OE defines a set of services:

- Naming Service: it is a CORBA Naming Service and must be present in the OE.
- Log Service: the OE may include a log service. If a log service is implemented, the log service shall conform to the OMG Lightweight Log Service Specification.
- Event Service: it is a CORBA Event Service and must be present in the OE.

Naming Service

The service is used to retrieve DomainManager and application components object references.

Static Stringified IORs are not allowed for application components: would not work for multiple instantiations of an application and Software Assembly Descriptor (SAD) files would not be portable.

The SCA defines a subset of the OMG Naming Service IDL that a Naming Service implementation must provide to be SCA compliant.

The minimum set of operations for Naming Service is based upon the operations needed by the ApplicationFactory for obtaining component's object references, application components for registering their object references, and the Application components to destroy naming context and component object references.

Naming Service Specifications

The OE shall provide an implementation of a CORBA Naming Service which implements the `CosNaming` module `NamingContext` interface operations:

- `bind`
- `bind_new_context`
- `unbind`
- `destroy`,
- `resolve`

as defined in the Appendix A of *OMG Interoperable Naming Service Specification*.
The id-and-kind pair of the Naming Service's NameComponent structure is s.t. the `id` element contains a string value that uniquely identifies a NameComponent. The `kind` element contains the "" (null string).

Log Service

The Log Service, if present, shall conform to the OMG Lightweight Log Service Specifications *Document formal/05-02-02: v1.1*

A log producer is a CF component (e.g., *DomainManager, Application, ApplicationFactory, DeviceManager, Device*) or an application's CORBA capable component (e.g., *Resource, ResourceFactory*).

Log producers shall implement a configure property which is a *CF Properties type* with an id of "`PRODUCER_LOG_LEVEL`" and a value. The value contains all log levels that are enabled. A log producer shall only output log records enabled.

Log producers and CF components required to write log records shall operate normally in the absence of a log service.
Log producers shall use their component identifier attribute in the producerId field of the log record.

Event Service

The OE provides an implementation of the CORBA Event Service.

The Event Service implements the *PushConsumer* and *PushSupplier* interfaces of the `CosEventComm` module as described in OMG *Event Service Specification* using the IDL found there.
The `CosEventComm` CORBA Module is used by consumers for receiving events and by producers for generating events:

- A component (e.g., *Resource, DomainManager*, etc.) that consumes events shall implement the `CosEventComm::PushConsumer interface`.
- A component (e.g., *Resource, Device, DomainManager*, etc.) that produces events shall implement the `CosEventComm::PushSupplier` interface and use the `CosEventComm::PushConsumer` interface for generating the events.

Event Service: Standard Event Channels

The OE provides two standard event channels:

- Incoming Domain Management Channel: called "`ODM_Channel`"
- Outgoing Domain Management Channel: called "`IDM_Channel`"

The Incoming Domain Management event channel is used by components within the domain to generate events (e.g., Device state change event) that are consumed by domain management functions (e.g., ApplicationFactory, Application, DomainManager, etc.).

The Outgoing Domain Management Channel is used by domain clients (e.g., HCI) to receive events (e.g., additions or removals from the domain) generated from domain management functions (e.g., ApplicationFactory, Application, DomainManager, etc.).

Besides these two standard event channels, the OE allows other event channels to be set up by application developers.

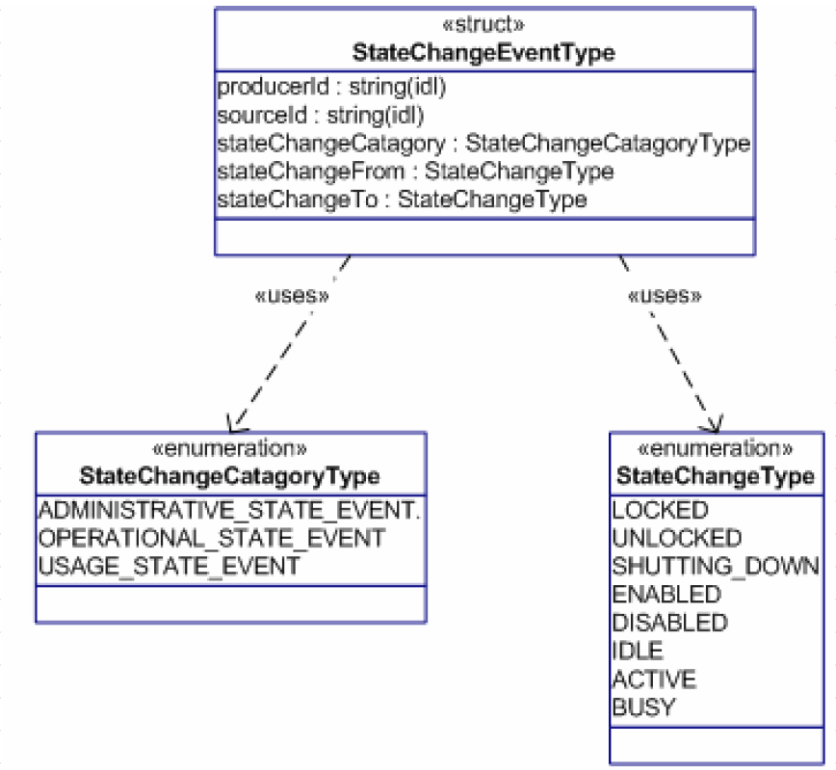
StandardEvent Module

The StandardEvent module contains type definitions that are used for passing events from event producers to event consumers (see, Appendix C of SCA specifications).

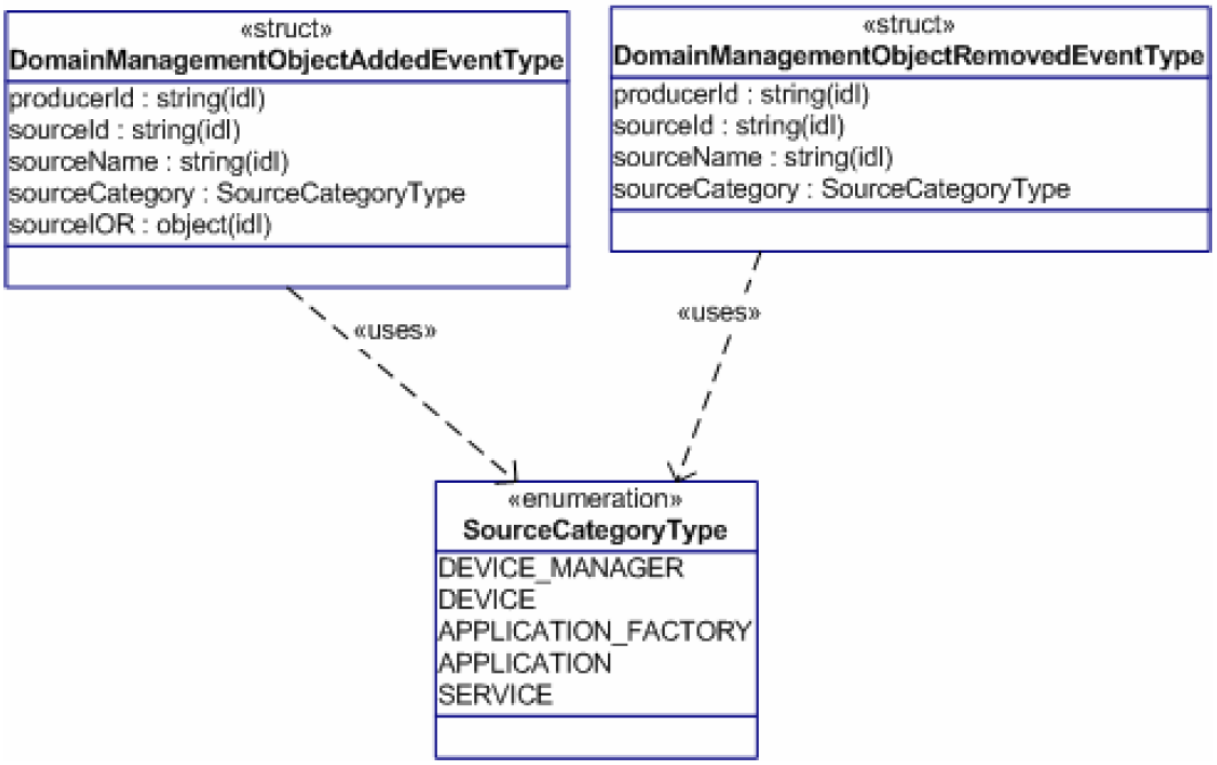
The defined event types are:

- StateChangeEventType: to indicate that the state of the event source has changed.
- DomainManagementObjectAddedEventType: to indicate that the event source has been added to the domain.
- DomainManagementObjectRemovedEventType: indicate that the event source has been removed from the domain.

StateChangeEventType



DomainManagementObjectAddedEventType and DomainManagementObjectRemovedEventType



Events: example

The *create* operation of an `ApplicationFactory` shall send a `DomainManagementObjectAddedEventType` event to the Outgoing Domain Management event channel upon successful creation of an application.

For this event:

1. The `producerId`: identifier attribute of the application factory.
2. The `sourceId`: identifier attribute of the created application.
3. The `sourceName`: name attribute of the created application.
4. The `sourceIOR`: object reference for the created application.
5. The `sourceCategory` is "`APPLICATION`".

Core Framework

As said, the Core Framework is the essential set of open application-layer CORBA interfaces.

The Core Framework consists of:

Base Application Interfaces: *Port, LifeCycle, TestableObject, PropertySet, PortSupplier, Resource*, and *ResourceFactory*;

Base Device Interfaces: *Device, LoadableDevice, ExecutableDevice*, and *AggregateDevice*;

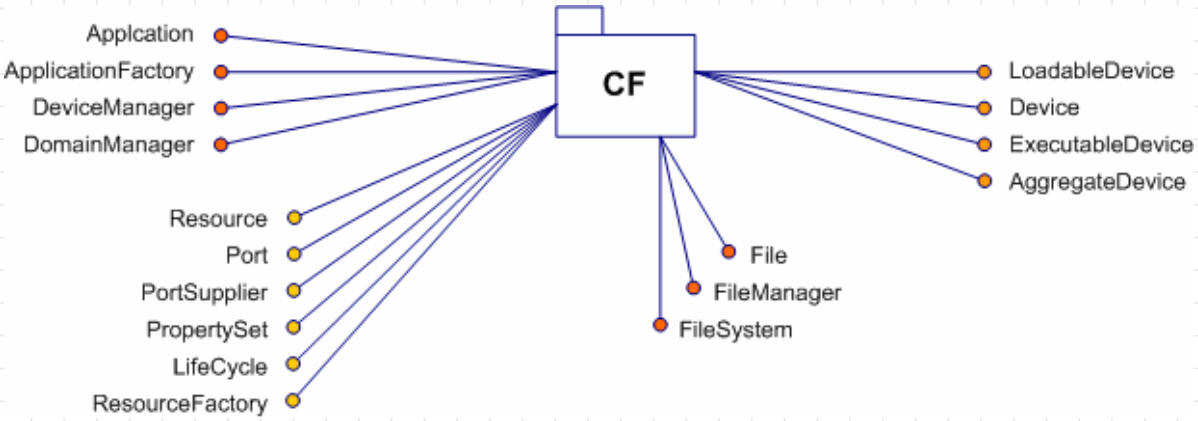
Framework Control Interfaces: *Application, ApplicationFactory, DomainManager*, and *DeviceManager*;

Framework Services Interfaces: *File, FileSystem*, and *FileManager*.

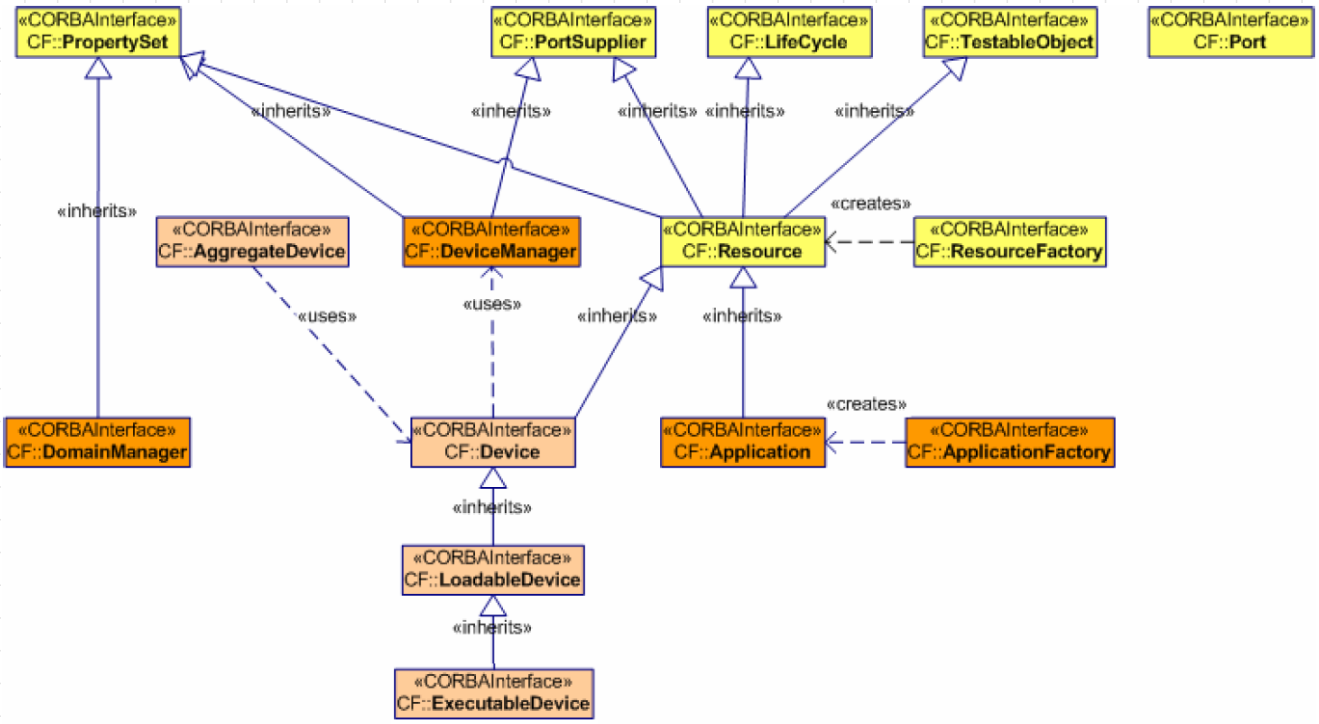
Core Framework CORBA Module

The CF interfaces are expressed in CORBA IDL: any IDL compiler for the target language of choice may compile the generated IDL.

The CF interfaces are contained in the CF CORBA module (see Appendix C of SCA specifications).



Core Famework IDL Relationship (key elements)



Core Framework in a nutshell...

A *DomainManager* component manages the software applications, application factories, hardware devices (represented by software devices) and device managers within the system.

Logical devices are software components that directly control the system's internal hardware devices: they implement the *Device*, LoadableDevice, or *ExecutableDevice* interfaces.

Other software components have no direct relationship with a hardware device, but perform application services for the user and implement the *Resource* interface. This interface provides a consistent way of configuring and tearing down these components.

Each resource can potentially communicate with other resources.

Core Framework in a nutshell.

An application is a specific collection of one or more resources which provides a specified service or function and which is managed through the *Application* interface.

The resources of an application are allocated to one or more hardware devices by the Application Factory based upon various factors including the current availability of hardware devices, the behavior rules of a resource, and the loading requirements of each resource.

The resources may be created by using the *ResourceFactory* interface and connected to other resources resident on the system.

Base Application Interfaces

We will start the description of the Core Framework from the Base Application Interfaces:

- *Port*
- *PortSupplier*
- *LifeCycle*
- *TestableObject*
- *PropertySet*
- *Resource*
- *ResourceFactory*

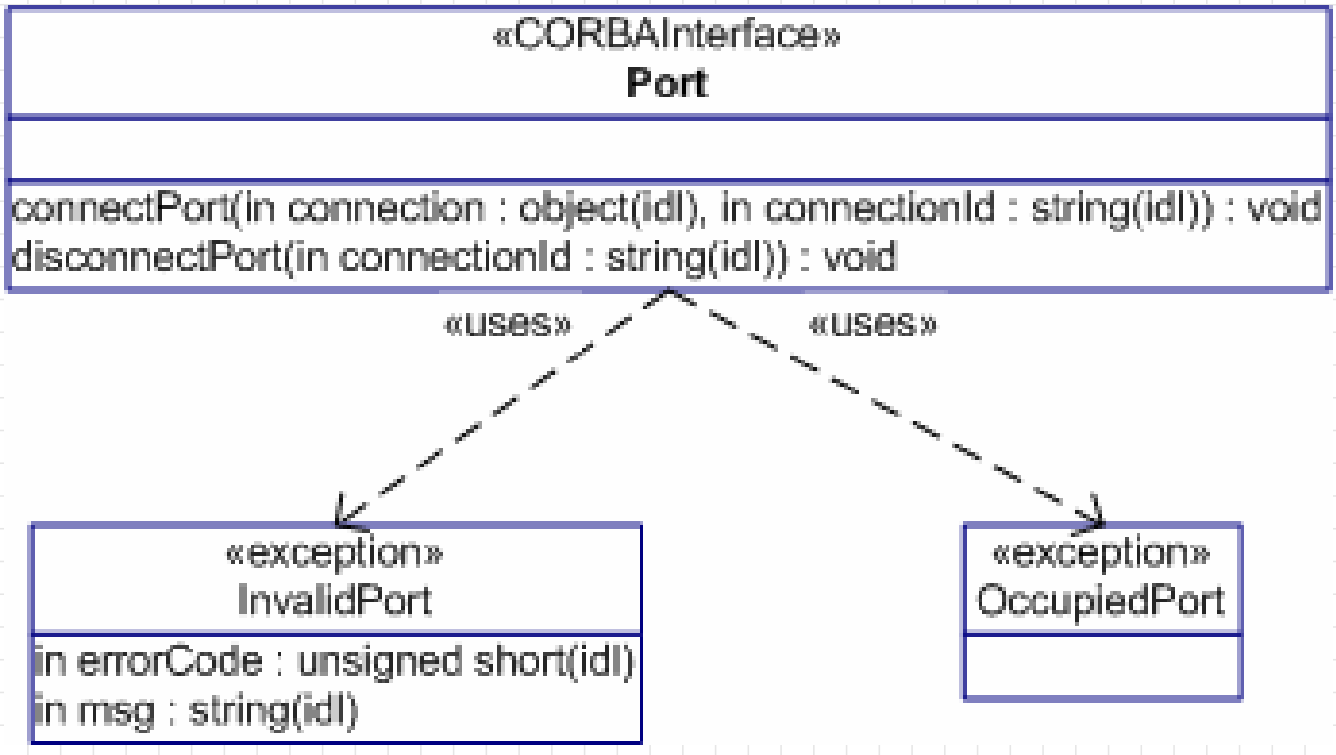
Base Application Interfaces: *Port*

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*.

In SCA this mechanism is supported by the *Port* interface.

Port: UML



Base Application Interfaces: *Port*

Application developers implement the *Port* interface for their uses port.

An application uses port must be a `Port Type`.

The uses and provides ports are paired up in the Software Profile's Software Assembly Descriptor (SAD) or DCD.

The provided generic operations allows *ApplicationFactory* and *Application* implementations to setup or tear down connections between any Application's CORBA software components.

Port Operations

```
void connectPort (in Object connection, in string
connectionId) raises (InvalidPort, OccupiedPort);
```

Applications require the `connectPort` operation to establish associations between ports. This operation shall make a connection to the component (a CORBA object reference) identified by its input parameters.

Provides half of a two-way association; two calls required for a two-way association.

A port may support several connections.

The input `connectionId` is a unique identifier to be used by the disconnectPort operation when breaking a specific connection.

`InvalidPort` exception: the input `connection` parameter is an invalid connection for this port.
`OccupiedPort` exception: the port is already fully occupied.

Port Operations

```
void disconnectPort (in string connectionId) raises
(InvalidPort);
```

The `disconnectPort` operation shall break the connection to the component identified by the input `connectionId` parameter.

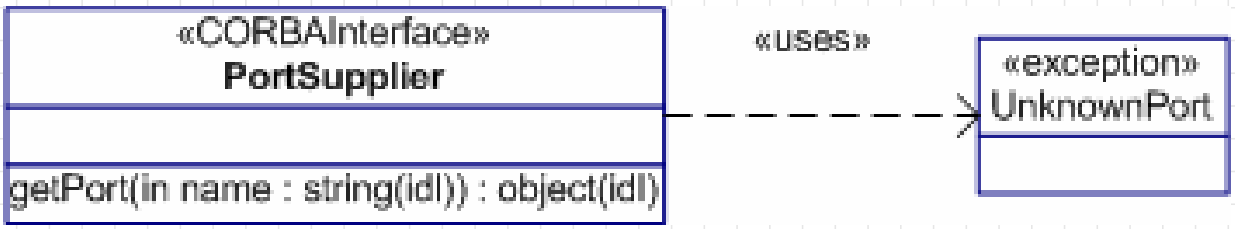
The `connectionId` parameter, for the Port operations, is a unique connection identifier created by the ApplicationFactory at the time a connection is created between uses and provides port or is the connection interface's connection ID in the SAD, if specified.

This supports generic fan-in and fan-out implementations without the uses or provides ports actually knowing the specific ports to which they are connected.

`InvalidPort` exception: the input `connectionId` parameter is not a known connection.

Base Application Interfaces: *PortSupplier*

This interface provides the `getPort` operation for those components that provide ports. (*Application, Resource, Device, DeviceManager*) as described in their SCD XML file.



PortSupplier Operations

```
Object getPort (in string name) raises (UnknownPort);
```

The `getPort operation` provides a mechanism to obtain a specific consumer or producer port.

A port supplier may contain zero-to-many consumer and producer port components, as specified in the component's software profile SCD.

The `getPort operation` is used by the ApplicationFactory and DomainManager to retrieve provides ports, in order to establish connections to services or to other components.

It returns the CORBA object reference to the named port as stated in the component's SCD.

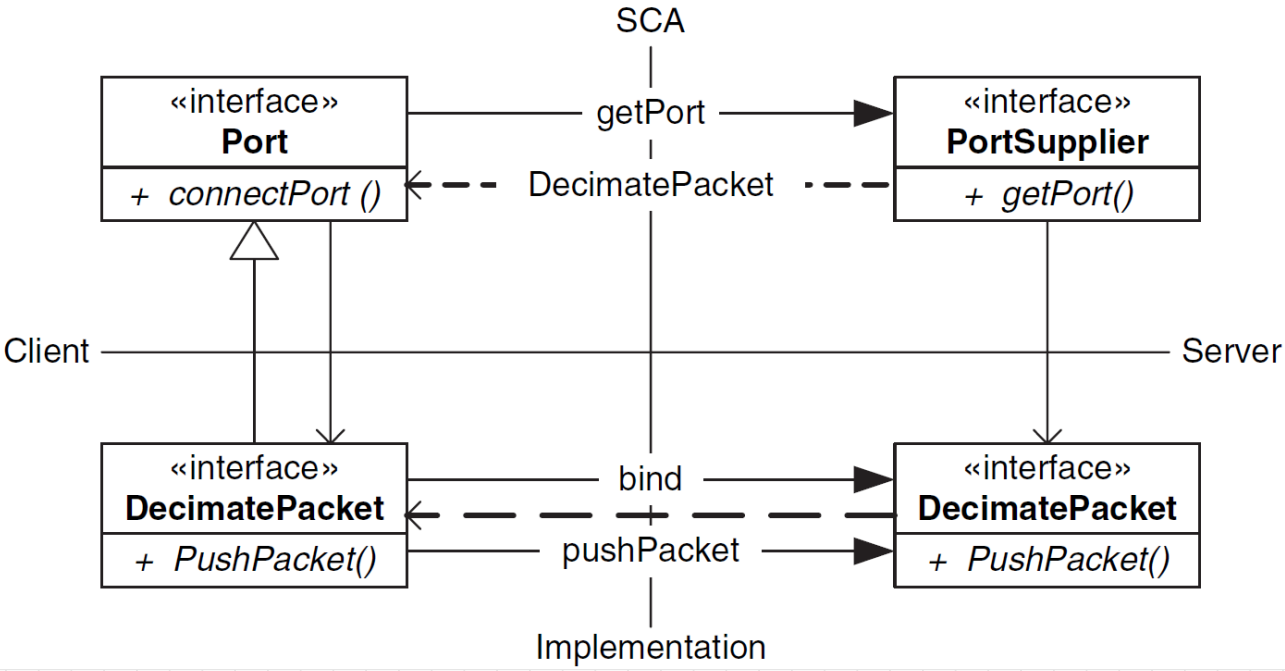
Implementation of Port Connections: an Example

The *Port* interface in the SCA is a means of obtaining a reference to the actual interface implemented by a component.

The interface implemented by the component provides the actual interface for performing the data transfer and control operations. Specific formats, data level protocols, and any other data structure or interpretation is implemented as part of the operational interface.

Thus, the Port interface is used merely for establishing a connection between two operational interfaces.

Implementation of Port Connections: an Example



Implementation of Port Connections: an Example

CLIENT

1. asks a reference to a server interface implementing one or more operations defined by the IDL through the `PortSupplier::getPort`.
2. the `connectPort is called` with the Port reference as an argument.
3. The connectPort implementation binds to the server endpoint implementing the PushPacket interface.
4. provides a sequence of packets.

SERVER

1. the `DecimatePacket` interface object reference is returned as a Port object.
2. receives the signal processing packet stream through the decimation filter.

Base Application Interfaces: *LifeCycle*

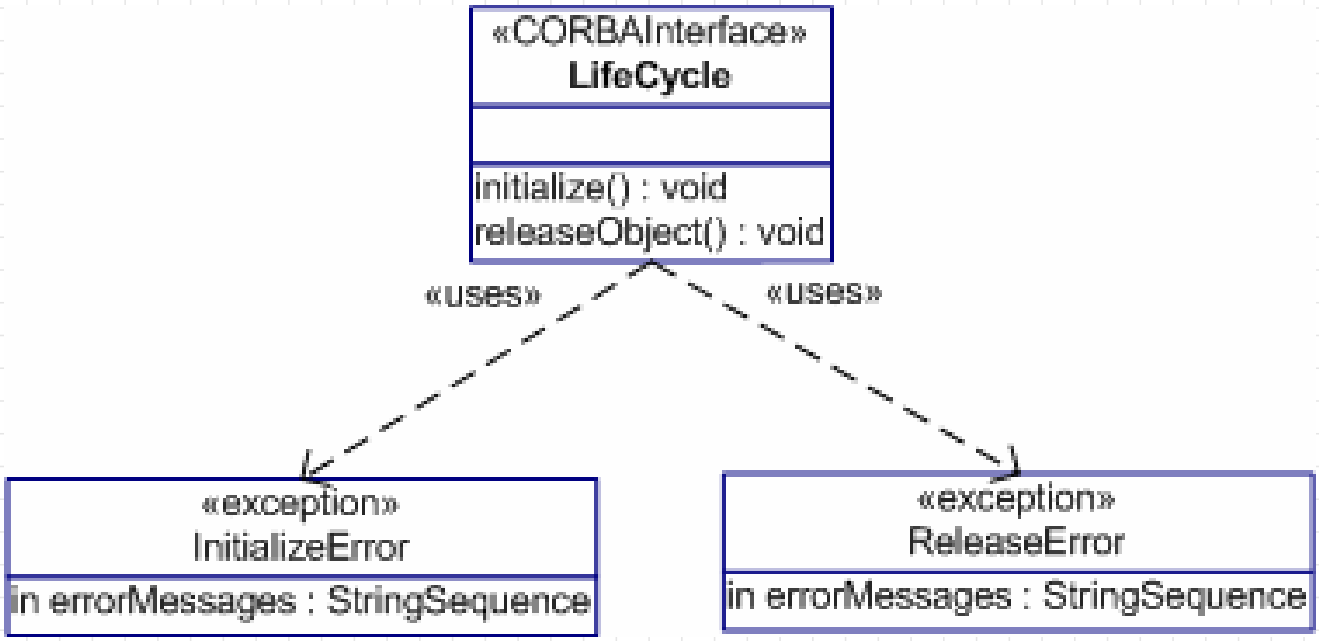
The *LifeCycle* interface defines the generic operations for initializing or releasing instantiated component-specific data and/or processing elements, like resources, applications, or devices within the domain.

ApplicationFactory implementations use this interface to initialize an application's components after components have been deployed and their object references have been obtained.

HCI (or other domain management clients) uses this interface to release an application or a device.

Application implementations use this operation to release an application's components.

LifeCycle UML



LifeCycle Operations

```
void initialize() raises (InitializeError);
```

The purpose is to provide a mechanism to set a component to a known initial state. For example, data structures may be set to initial values, memory may be allocated, hardware devices may be configured to some state, etc.

```
void releaseObject() raises (ReleaseError);
```

The purpose of the releaseObject operation is to provide a means by which an instantiated component may be torn down.

Base Application Interfaces: *TestableObject*

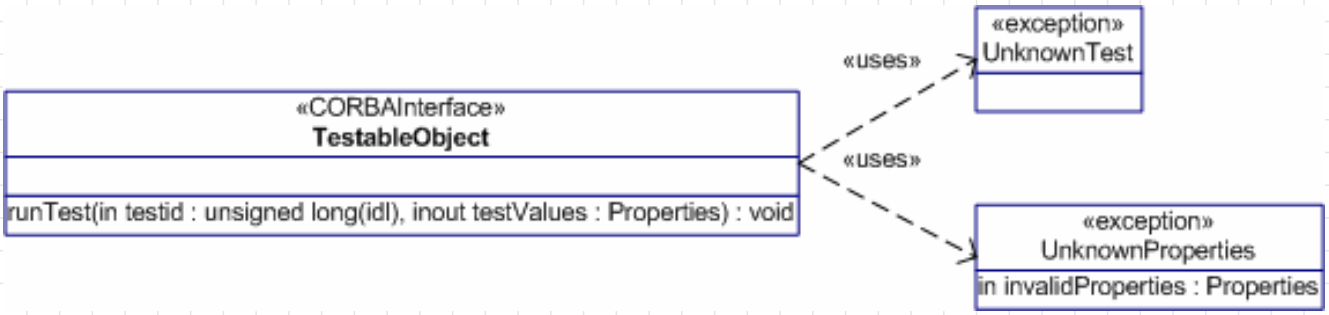
The *TestableObject* interface is a generic way of testing a *Resource, Application, or Device* components within a domain.

Test descriptions, along with their results, are described in the Software Profile's Properties File by the *test XML element*.

TestableObject may be used by a generic HCI to test an Application or Device components within the domain.

TestableObject can be used for testing remotely over the air without operator intervention.

TestableObject UML



Used types:

```
struct DataType
  {
    string id;

    any value;
  };

typedef sequence <DataType> Properties;
```


TestableObject Operation

`void runTest (in unsigned long testId, inout Properties testValues) raises (UnknownTest, UnknownProperties);`

The input `testId` parameter is used to determine which of its predefined test implementations should be performed.

The id/value pair(s) of the `testValues` parameter are used to provide additional information to the implementation-specific test to be run.

The `runTest` operation shall return the result(s) of the test in the `testValues` parameter.

Valid `testId` and both input and output `testValues` (properties) shall at a minimum be the test blue properties defined in the properties test element of the component's Properties Descriptor.

The `runTest operation` shall not execute any testing when the input `testId` or any of the input `testValues` are not known by the component or are out of range.

Base Application Interfaces: *PropertySet*

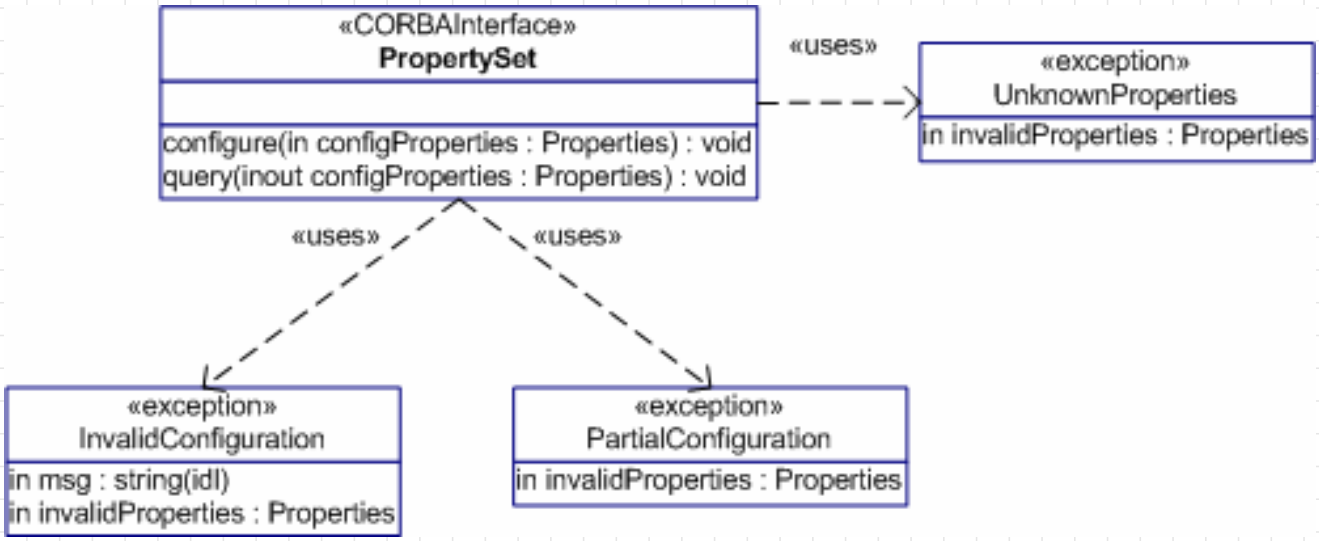
The *PropertySet* interface defines `configure` and `query` operations to access component properties/attributes.

The `PropertySet` interface is used to support the *properties* element. Properties (Id(name)/value pairs) are concepts from the CORBA Components, Telecommunications Information Networking Architecture (TINA) specifications, and CORBA property service specification.

ApplicationFactory implementations use this interface to initially configure an application.

PropertySet can be used by a generic HCI to configure an *Application*, *Device*, *DeviceManager*, and *DomainManager* within the domain. *PropertySet* can be used for over the air remote configuration or query without operator intervention.

PropertySet UML



PropertySet Operations

`void configure (in Properties configProperties) raises (InvalidConfiguration, PartialConfiguration);`

The `configure` operation allows id/value pair configuration properties to be assigned to components implementing this interface.

It assigns values to the properties as indicated in the input `configProperties` parameter.

Valid properties for the configure operation shall at a minimum be the configure readwrite and writeonly properties referenced in the component's SPD.

`InvalidConfiguration` exception: when a configuration error occurs and no configuration properties were successfully set.

`PartialConfiguration` exception: some configuration properties were successfully set and some configuration properties were not successfully set.

PropertySet Operations

`void query (inout Properties configProperties) raises (UnknownProperties);`

The `query` operation allows a component to be queried to retrieve its properties.

It returns *all* component properties when the inout parameter `configProperties` is zero size.

It returns *only* the id/value pairs specified in the `configProperties` parameter if it is not zero size.

Valid properties for the query operation shall be all configure properties as referenced in the component's SPD.

`UnknownProperties` exception: when one or more properties being requested are not known by the component.

Base Application Interfaces: *Resource*

The *Resource* interface provides a common API to initialize, configure, and control a software component (e.g., a Device, or an Application).

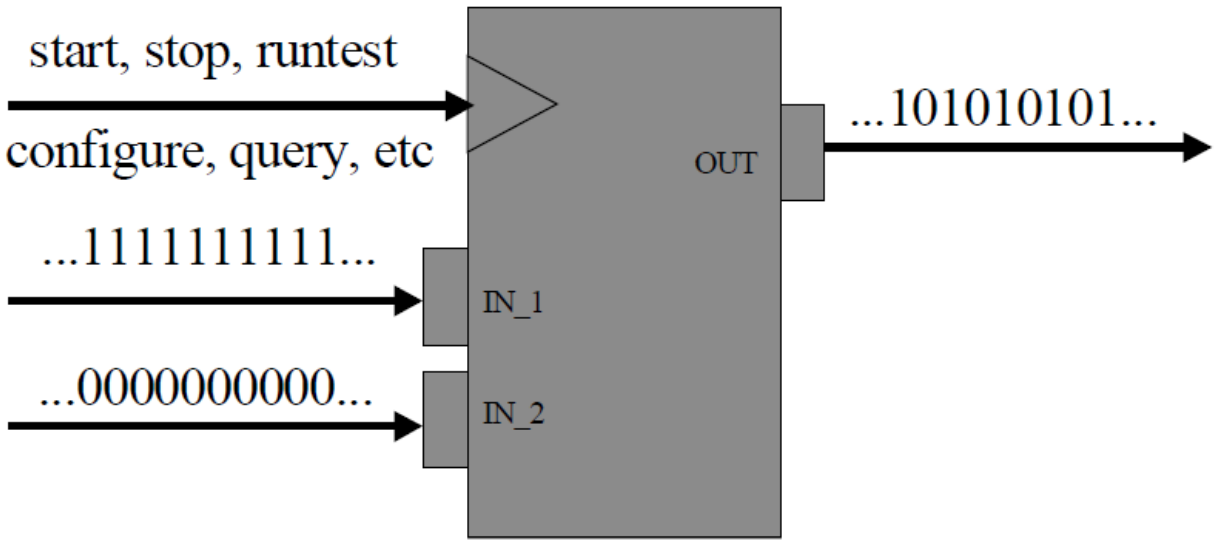
The *Resource* interface inherits from the *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interfaces.

The *Resource* interface may also be inherited by other application interfaces.

Resource could be used by a generic HCI (or other domain management clients) to configure an Application or Device within the domain.

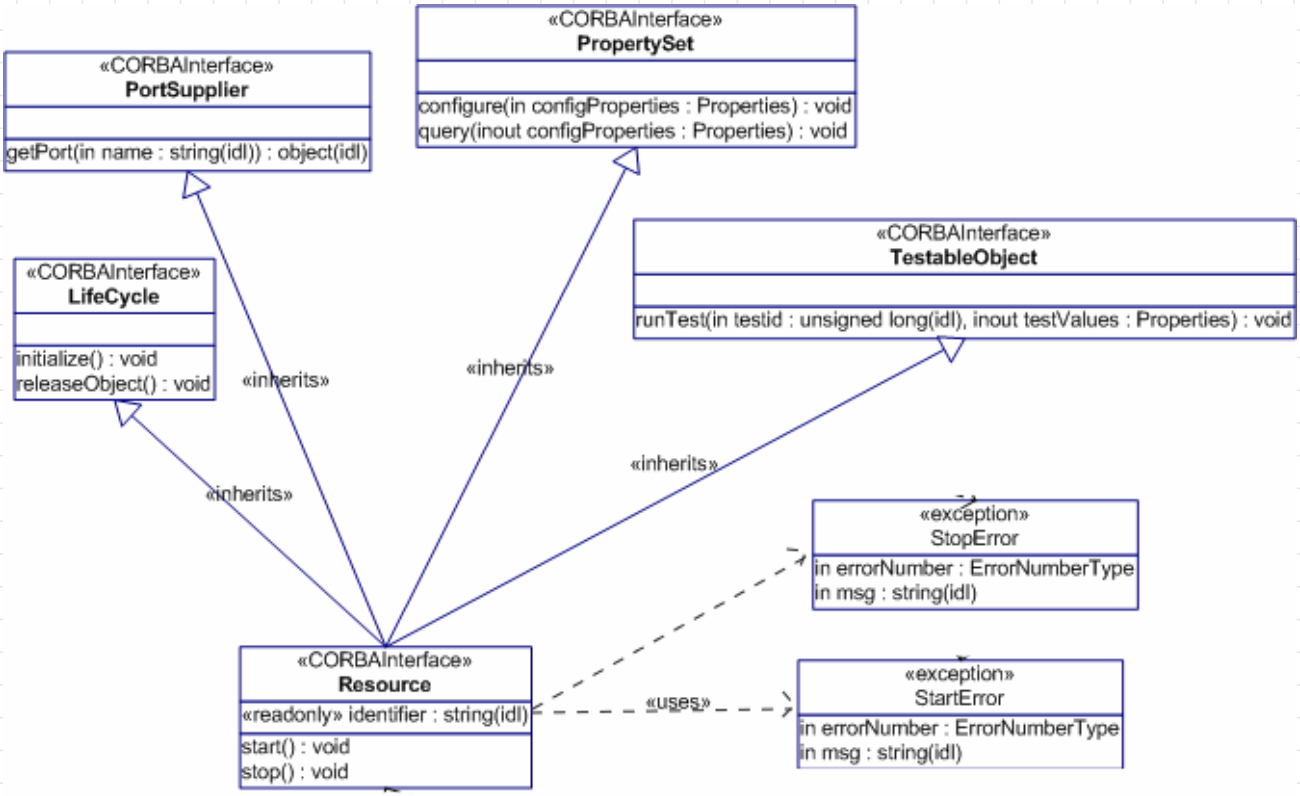
Each *Resource* in the system has a unique identifier so that each component can be identified (e.g., as the producer of a log message).

Resource example



A multiplexer *Resource* with two data ports and one data output port.

Resource UML



Resource Operations

```
void start() raises (StartError);
```

The start operation is provided to command the resource implementing this interface to start internal processing.

StartError exception: when an error occours in starting the component. Errors starting with "CF_E" map POSIX errors defined in *POSIX Realtime Application Support, IEEE Std 1003.13-2003*.

```
enum ErrorNumberType { CF_NOTSET, CF_E2BIG, CF_EACCES, CF_EAGAIN,
CF_EBADF, CF_EBADMSG, CF_EBUSY, CF_ECANCELED, CF_ECHILD, CF_EDEADLK,
CF_EDOM, CF_EEXIST, CF_EFAULT, CF_EFBIG, CF_EINPROGRESS, CF_EINTR,
CF_EINVAL, CF_EIO, CF_EISDIR, CF_EMFILE, CF_EMLINK, CF_EMSGSIZE,
CF_ENAMETOOLONG, CF_ENFILE, CF_ENODEV, CF_ENOENT, CF_ENOEXEC, CF_ENOLCK,
CF_ENOMEM, CF_ENOSPC, CF_ENOSYS, CF_ENOTDIR, CF_ENOTEMPTY, CF_ENOTSUP,
CF_ENOTTY, CF_ENXIO, CF_EPERM, CF_EPIPE, CF_ERANGE, CF_EROFS, CF_ESPIPE,
CF_ESRCH, CF_ETIMEDOUT ,CF_EXDEV };

exception StartError { ErrorNumberType errorNumber; string msg; };
```

CF_NOTSET is an SCA specific value applicable for any exception when the POSIX error values are not appropriate.

Resource Operations

`void stop() raises (StopError);`

The stop operation is provided to command the resource implementing this interface to stop internal processing.

It shall not inhibit subsequent configure, query, and start operations.

StopError exception: when an error occours during an attempt to stop the resource component.

```
exception StartError { ErrorNumberType errorNumber; string msg; };
```


Base Application Interfaces: *ResourceFactory*

A resource factory is used to create and tear down a resource. It follows the Factory Design Patterns.

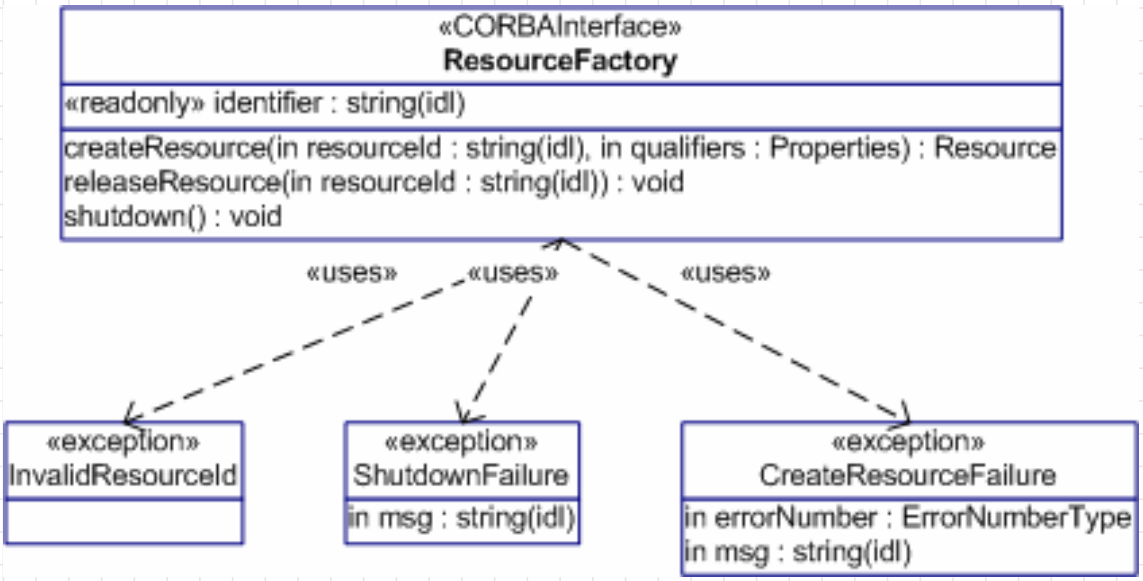
The factory mechanism provides client-server isolation among resources and provides a standard mechanism of obtaining a resource without knowing its identity.

An application is not required to use resource factories, but, if used, a SPD specifies which *Resource* factories are to be used by the Application factory

The *ResourceFactory* keeps track of the number of times the *Resource* has been referenced by clients using the Resource.

As clients release their reference to the *Resource* the factory destroys the *Resource* when there are no more references from any client.

ResourceFactory UML



ResourceFactory Operations

```
Resource createResource (in string resourceId, in Properties
qualifiers) raises (CreateResourceFailure);
```

The `createResource` operation provides the capability to create resources in the same process space as the resource factory or to return a reference to a resource already created.

The `resourceId` parameter is the identifier for a resource.

The `qualifiers` parameter contains values used by the resource factory in creation of the Resource.

The `qualifiers` may be used to identify the type of Resource to be created. It is ignored if the resource already exists for the given resourceId.

A counter is used to track the number of references to a resource. The resource is not released if the couter is not zero.

ResourceFactory Operations

```
void releaseResource (in string resourceId) raises
(InvalidResourceId);
```

The releaseResource operation decrements the reference count for the specified resource indicated by the resourceId parameter.

If the resource's reference count is zero, the releaseResource operation releases the resource from the CORBA environment (server side) and make the resource no longer available.

ResourceFactory Operations

```
void shutdown() raises (ShutdownFailure);
```

The shutdown operation provides the mechanism for releasing the resource factory from the CORBA environment (server side).

ShutdownFailure exception: when processing errors prevent the release of the resource factory from the CORBA environment or when all resources have not been released from the resource factory.

Base Device Interfaces

We will proceed with the description of the Core Framework by discussing the Base Device Interfaces:

- *Device*
- *LoadableDevice*
- *ExecutableDevice*
- *AggregateDevice*

Devices in SCA

A core ability of the SCA is to represent and manage the underlying physical hardware that implements the radio system.

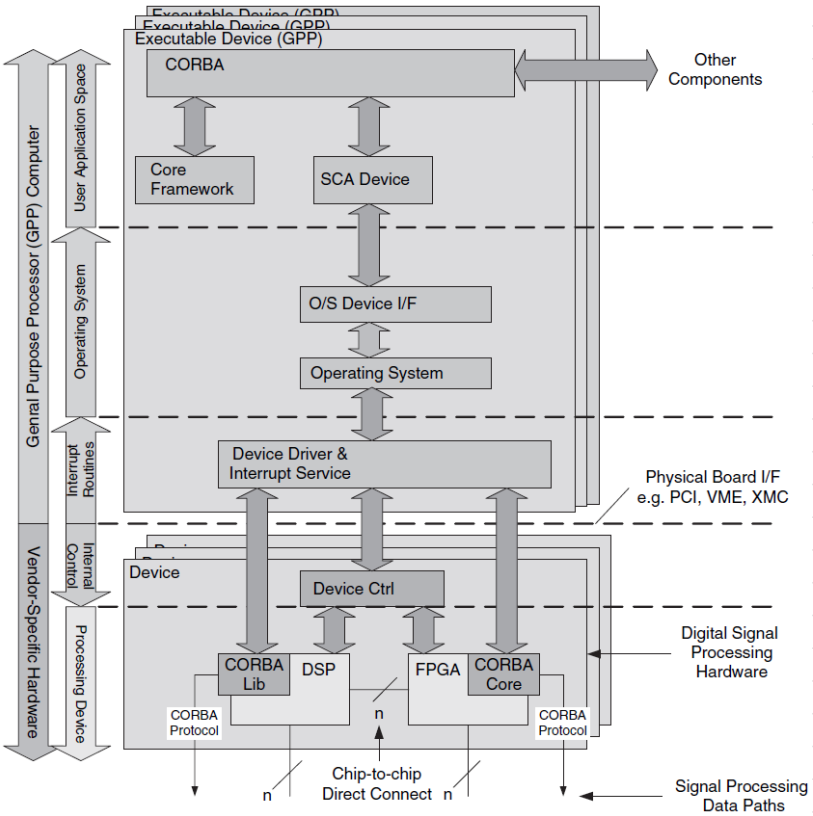
The approach within the SCA is to define a minimal set of interfaces that provide essential management and control capabilities for all devices within the radio system.

In the context of an SCA radio system, an SCA Device is a logical interface to the underlying physical hardware. This hardware includes any physical component that processes any part of the signal chain from the antenna through to the I/O connection.

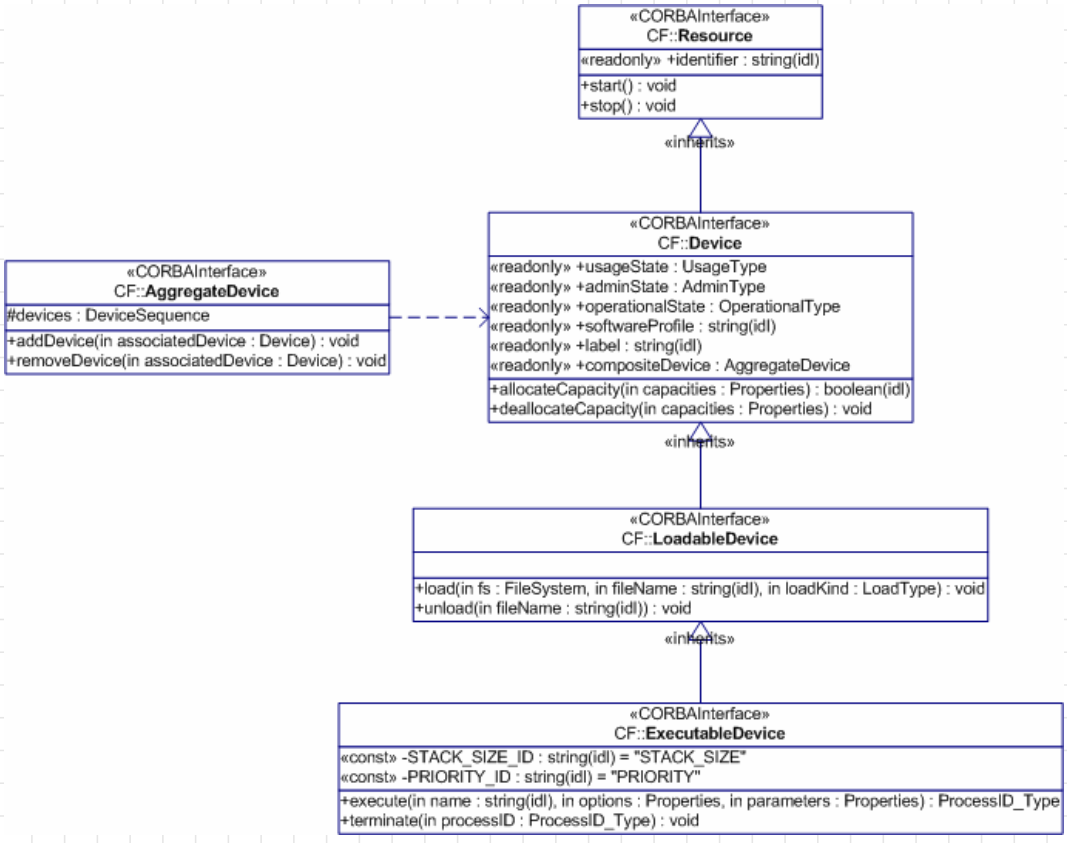
The SCA *Device* implementation resides at the application level.

In general, *Device* interface does not cover all the API calls to the device as provided by the manufacturer. If there are extensions to be made available to SCA application components, then the *Device* interface would be extended by deriving another interface class from the Device interface.

Layering in an SCA Device interface



Logical Device Interface Relationships



Base Device Interfaces: *Device*

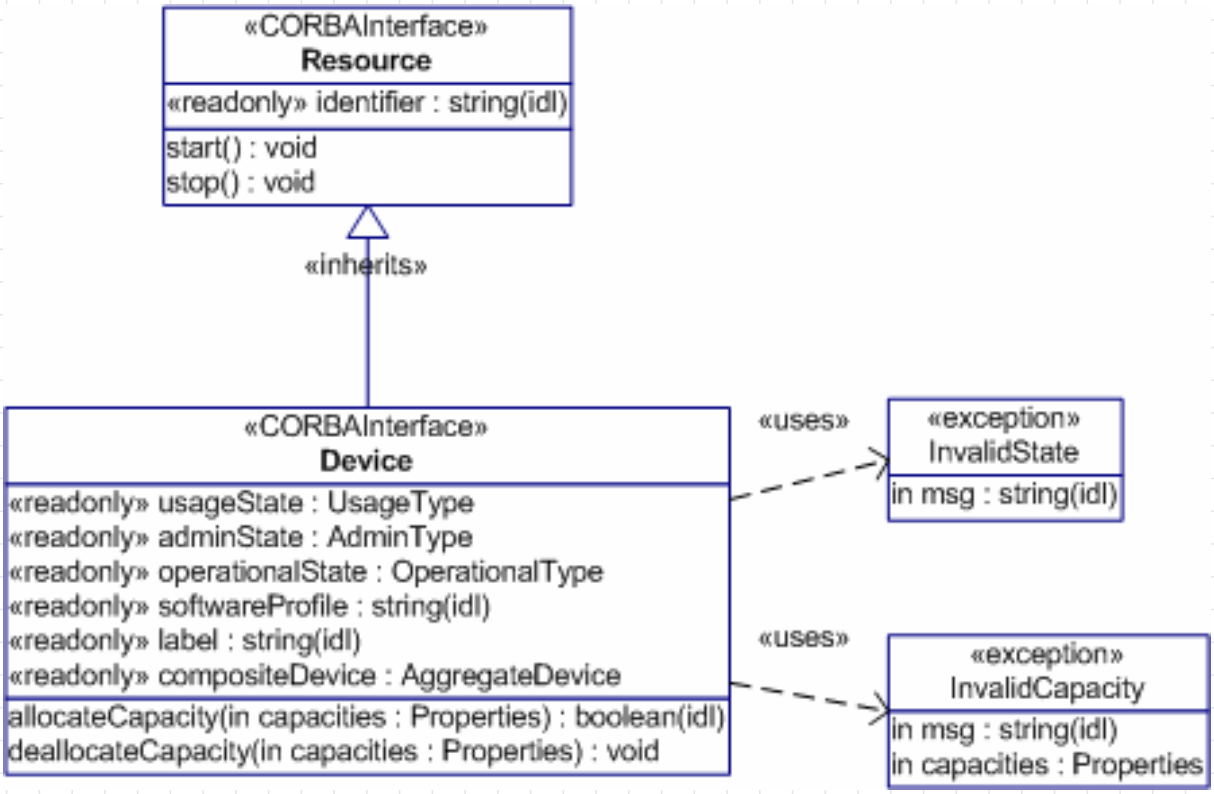
A *Device* is a type of *Resource* interface.

The *Device* interface defines additional capabilities and attributes for any logical device in the domain.

It provides attributes and operations:

1. Software Profile Attribute — The SPD referenced by this profile element (Profile Descriptor) defines the logical device capabilities (data/command uses and provides ports, configure and query properties, capacity properties, status properties, etc.), which could be a subset of the hardware device's capabilities.
2. State Management & Status Attributes -– This information describes the administrative, usage, and operational states of the device.
3. Capacity Operations – In order to use a device, certain capacities (e.g., memory, performance, etc.) are obtained from the device. A device may have multiple capacities which need to be allocated.

Device UML



Device Attributes

`readonly attribute UsageType usageState;`

The readonly usageState indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for allocation at that instant.

```
enum UsageType
  {
    IDLE,
    ACTIVE,
    BUSY
  };
```

IDLE – not in use

ACTIVE – in use, with capacity remaining for allocation

BUSY – in use, with no capacity remaining for allocation

Device Attributes

`attribute AdminType adminState;`

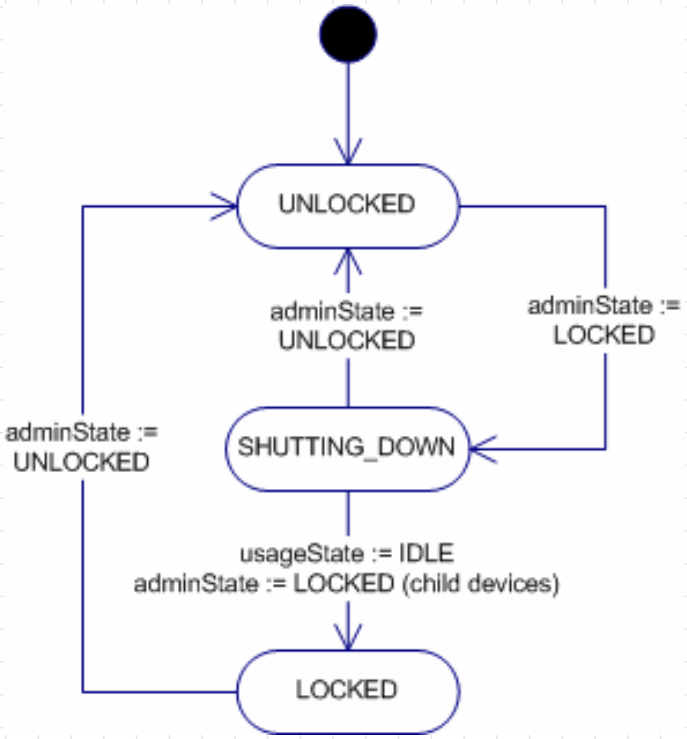
The adminState indicates indicates the permission to use or prohibition against using the device.

```
enum AdminType
  {
    LOCKED,
    SHUTTING_DOWN,
    UNLOCKED
  };
```

The adminState attribute shall only allow the setting of LOCKED and UNLOCKED values.

The adminState attribute, upon being commanded to be LOCKED, shall transition from UNLOCKED to SHUTTING_DOWN and set the adminState to LOCKED for its entire aggregation of devices (if any). The adminState shall then transition to LOCKED when the device's usageState is IDLE and its entire aggregation of devices are LOCKED.

Transition Diagram for adminState



Device Attributes

`readonly attribute OperationalType operationalState;`

The `operationalState` indicates whether or not the device is functioning.

```
enum OperationalType
  {
    ENABLED,
    DISABLED
  };
```

State Changes and Events: example

When a device changes its state, it shall generate an event.

E.g., the device shall send a `StateChangeEventType event` to the Incoming Domain Management (IDM) event channel, whenever the `usageState` attribute changes from `ACTIVE` to `BUSY`.

1. The `producerId`: identifier attribute of the device.
2. The `sourceId`: identifier attribute of the device.
3. The `stateChangeCategory`: is `USAGE_STATE_EVENT`.
4. The `stateChangeFrom`: is `ACTIVE`.
5. The `stateChangeTo`: is `BUSY`.

Device Attributes

```
readonly attribute string softwareProfile;
```

The `softwareProfile` attribute contains a profile element (Profile Descriptor) with a file reference to the SPD file.

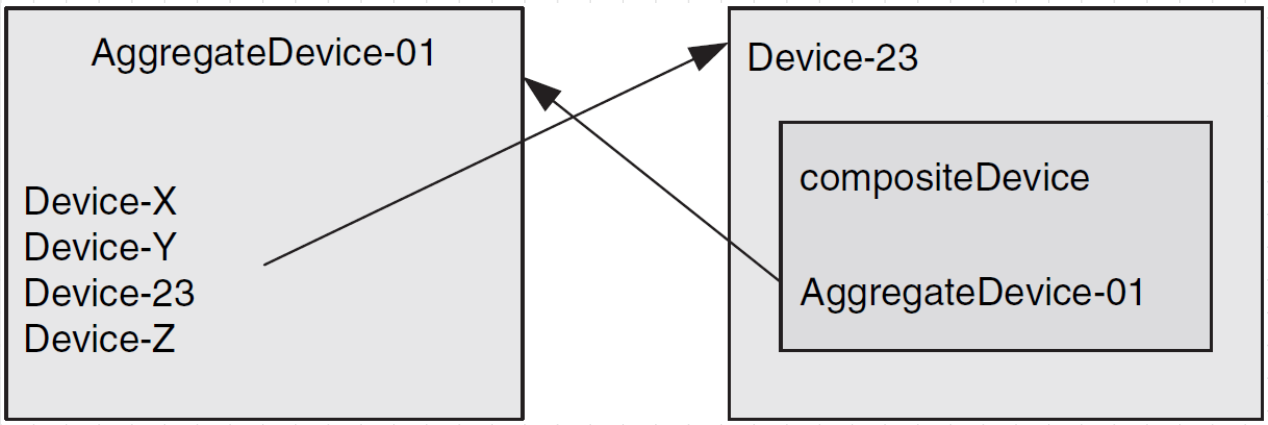
```
readonly attribute string label;
```

The `label` attribute contains the device's label: the meaningful name given to a device.

Device Attributes

```
readonly attribute AggregateDevice compositeDevice;
```

The compositeDevice attribute contains the object reference of the aggregate device, with which this Device is associated, or a nil CORBA object reference if no association exists.



Device Operations

```
boolean allocateCapacity (in Properties capacities) raises
(InvalidCapacity, InvalidState);
```

The allocateCapacity operation shall reduce the current capacities of the device based upon the input capacities parameter, when the adminState is UNLOCKED, the operationalState is ENABLED, and the usageState is not BUSY.

The allocateCapacity operation shall change the device's usageState attribute to BUSY, when the device determines that it is not possible to allocate any further capacity, otherwise to ACTIVE.

The allocateCapacity operation shall return TRUE, if the capacities have been allocated, or FALSE, if not allocated.

Device Operations

`void deallocateCapacity (in Properties capacities) raises (InvalidCapacity, InvalidState);`

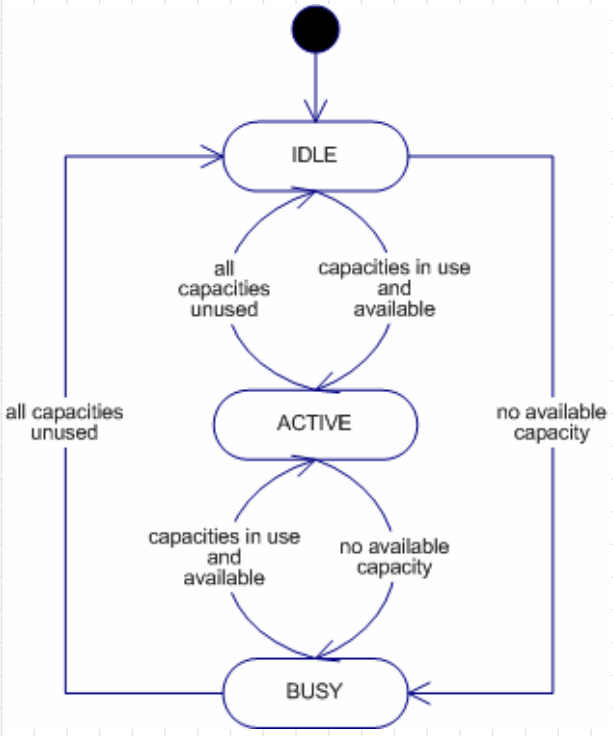
The `deallocateCapacity operation provides the mechanism to return capacities back` to the device, making them available to other users.

It operates upon the input capacities parameter.

The `deallocateCapacity operation shall set the usageState attribute to ACTIVE (IDLE, resp.)` when, after adjusting capacities, any (none, resp.) of the device's capacities are still being used.

`InvalidCapacity exception`: the capacity ID is invalid or the capacity value is wrong.
`InvalidState exception`: the device's state is `LOCKED` or `DISABLED`.

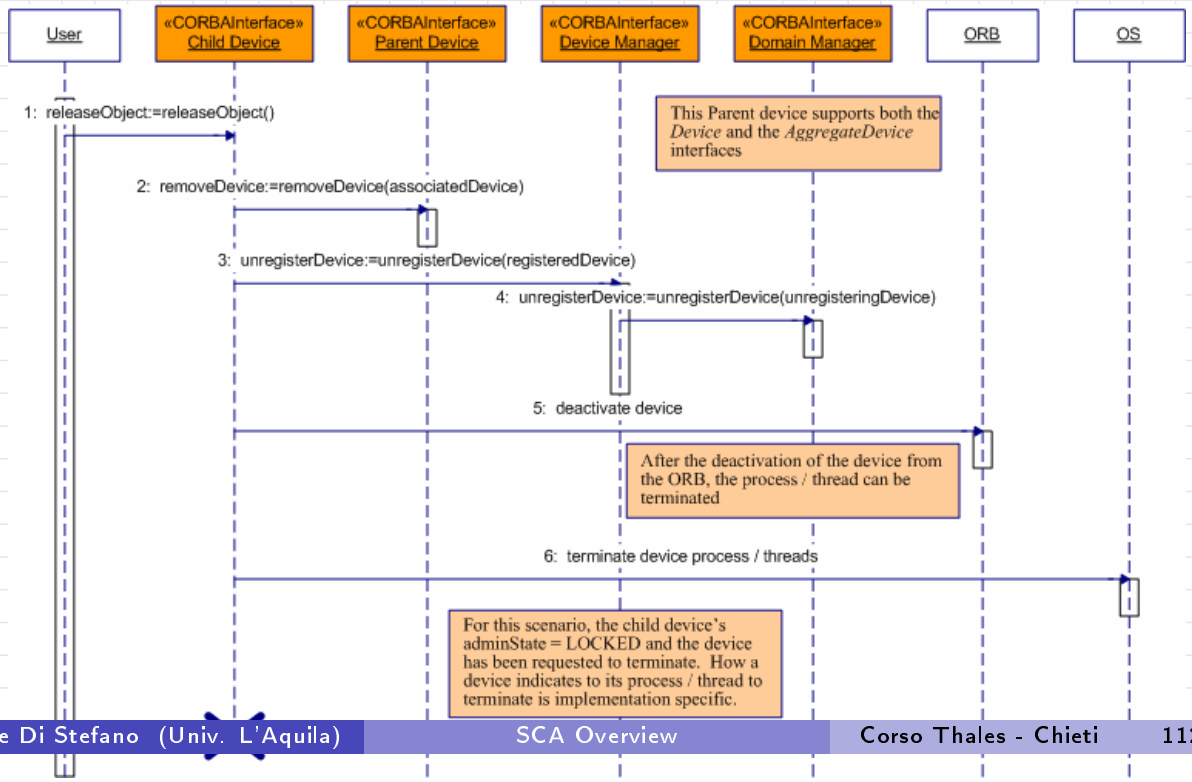
Transition Diagram for `allocateCapacity` and `deallocateCapacity`

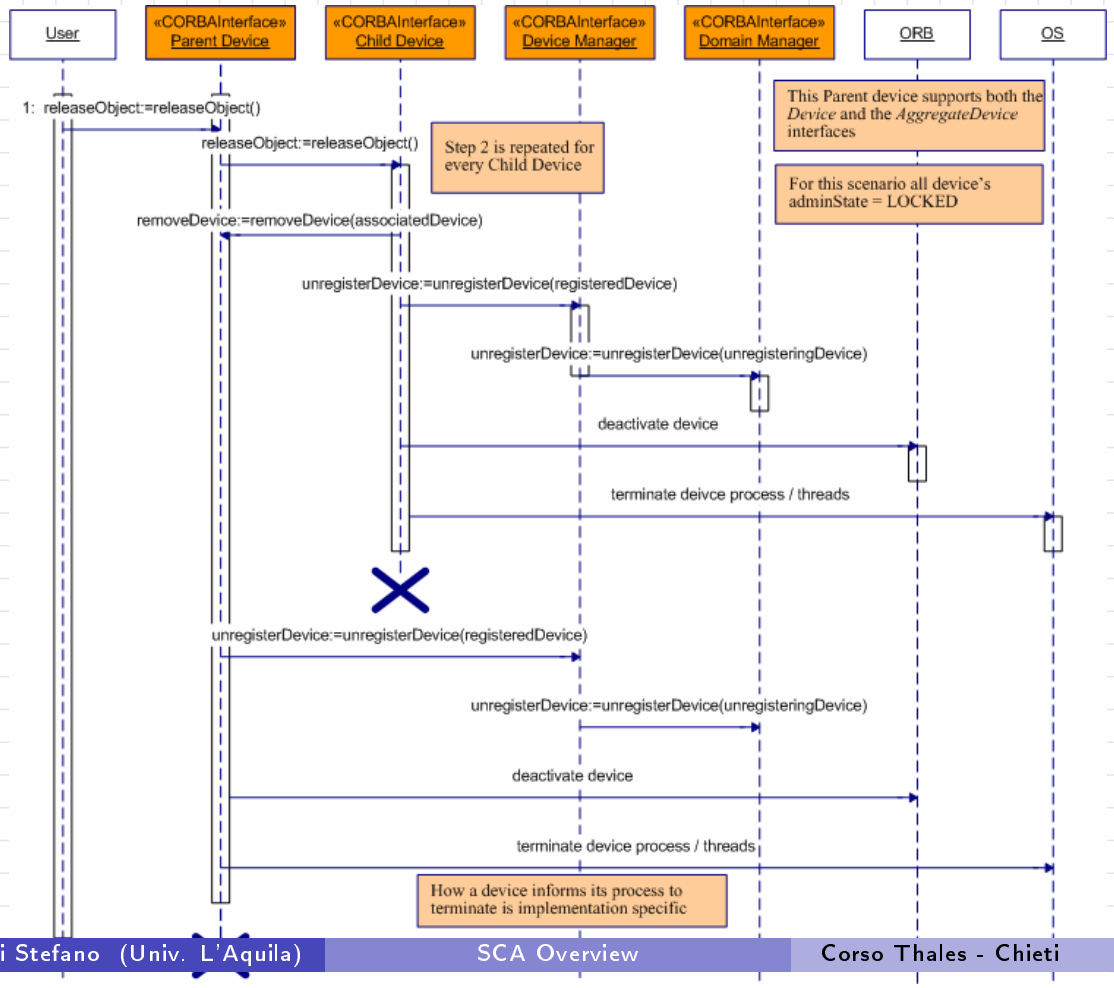


Device Operations

`void releaseObject() raises (ReleaseError);`

Additional behavior is required for the `LifeCycle::releaseObject`.





Base Device Interfaces: *LoadableDevice*

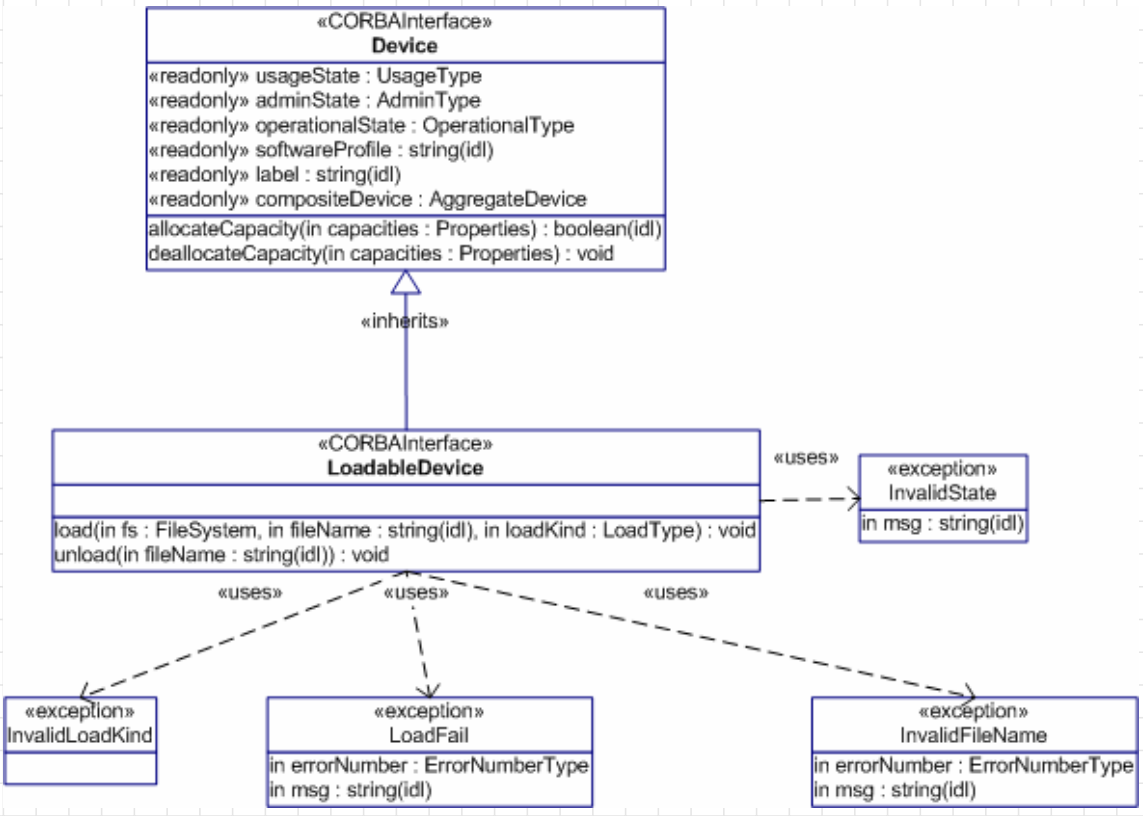
This interface extends the *Device* interface by adding software loading and unloading behavior to a device.

Some devices only support loadable behavior; this interface provides the mechanism for loading software onto these devices.

The actual implementation of this interface vary depending on the underlying OE.

There are different types of load (kernel module, driver, shared library, executable) that can be performed by the load operation. These types are based upon most OS capabilities.

LoadableDevice UML



LoadableDevice Operations

`void load (in FileSystem fs, in string fileName, in LoadType loadKind) raises (InvalidState, InvalidLoadKind, InvalidFileName, LoadFail);`

The load operation provides the mechanism for loading software on a device that may be subsequently executed, if it is an executable device.

The `fileName` parameter is a pathname relative to `FileSystem` parameter.

The load operation shall support the load types as stated in the device's software profile `LoadType` allocation properties.

Multiple loads of the same file are possible and the load operation should account for this so that the unload operation behavior can be performed.

```
enum LoadType
  {
    KERNEL_MODULE,
    DRIVER,
    SHARED_LIBRARY,
    EXECUTABLE
  };
```


LoadableDevice Operations

`void unload (in string fileName) raises (InvalidState, InvalidFileName);`

The unload operation provides the mechanism to unload software that is currently loaded.

The unload operation shall unload the file identified by the input `fileName` parameter from the device when the number of unload requests matches the number of load requests for the indicated file.

`InvalidState` exception: if the device is LOCKED or DISABLED.
`InvalidFileName` exception: the file designated by the input `fileName` parameter cannot be found.

Base Device Interfaces: *ExecutableDevice*

This interface extends the *LoadableDevice* interface by adding execute and terminate behavior to a device.

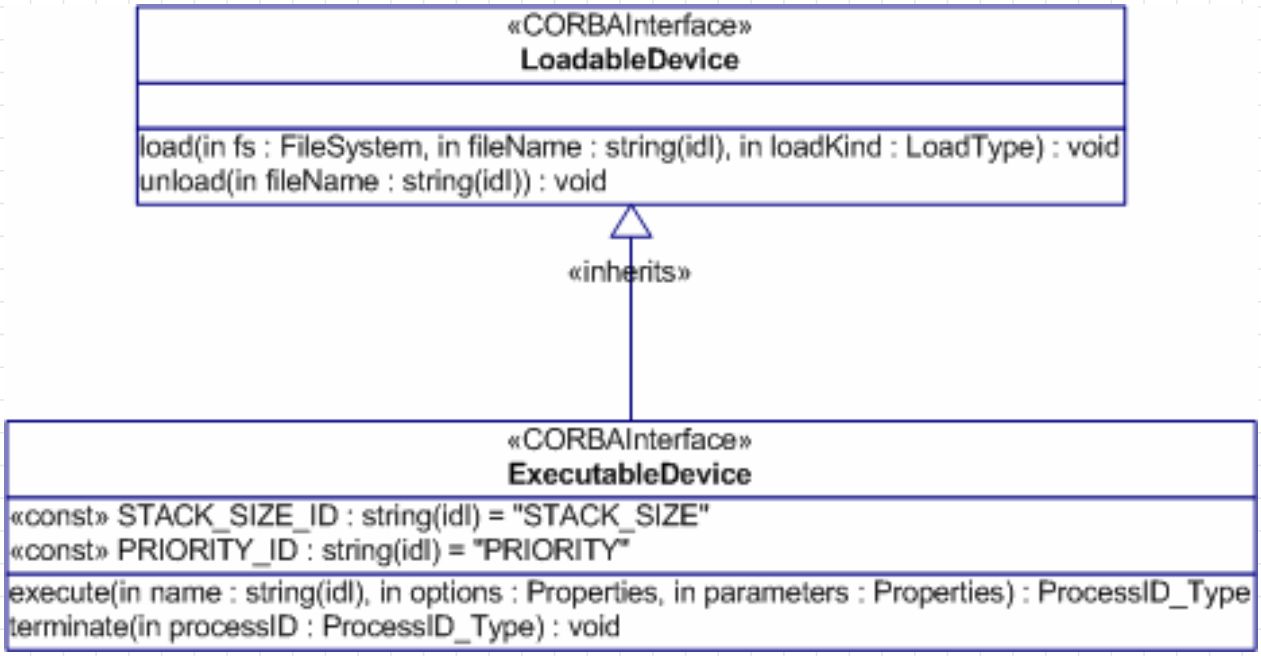
The *ExecutableDevice* interface is usually used for devices that have OS (e,g,, VxWorks, LynxWorks, Linux, etc.) that support creation of threads/processes.

The execute and terminate operations' implementation behavior vary depending on the underlying OE.

The parameters for the `execute` operation allow for the user to have control over the stack size (`STACK_SIZE`) and priority (`PRIORITY`) for the thread/process creation and for user parameters to be passed to the thread/process during creation.

The user parameters are id and value string pairs so they can be converted to (argc, argv) format, as used in POSIX.

ExecutableDevice UML



ExecutableDevice Operations

`ProcessID_Type execute (in string name, in Properties options, in Properties parameters) raises (InvalidState, InvalidFunction, InvalidParameters, InvalidOptions, InvalidFileName, ExecuteFail);`

The execute operation shall execute the function or file identified by the input name parameter using the input parameters and options parameters. Whether it is a function or a file is implementation-specific.

It shall convert the input parameters (id/value string pairs) parameter to the standard argv of the POSIX exec family of functions.

The execute operation input options parameters are STACK_SIZE_ID and PRIORITY_ID. They are used, when specified, to set the OS's process/thread stack size and priority, for the executable image.

The execute operation shall return a unique process ID for the process that it created.

execute exceptions

InvalidState exception: the device's is LOCKED, SHUTTING_DOWN or DISABLED.

InvalidFunction exception: the function indicated by the name parameter does not exist.

InvalidFileName exception: the file name indicated by the name parameter does not exist.

InvalidParameters exception: the input parameter ID or value attributes are not valid strings.

InvalidOptions exception: the input options parameter does not comply with STACK_SIZE_ID and PRIORITY_ID.

ExecuteFail exception: the operating system "execute" function for the device is not successful.

ExecutableDevice Operations

```
void terminate (in ProcessID_Type processId) raise
(InvalidProcess, InvalidState);
```

The `terminate` operation terminates the execution of a process/thread, designated by the `processId` input parameter, that was started up with the execute operation.

`InvalidState` exception: the device is `LOCKED` or `DISABLED`.
`InvalidProcess` exception: the `processId` does not exist.

Base Device Interfaces: *AggregateDevice*

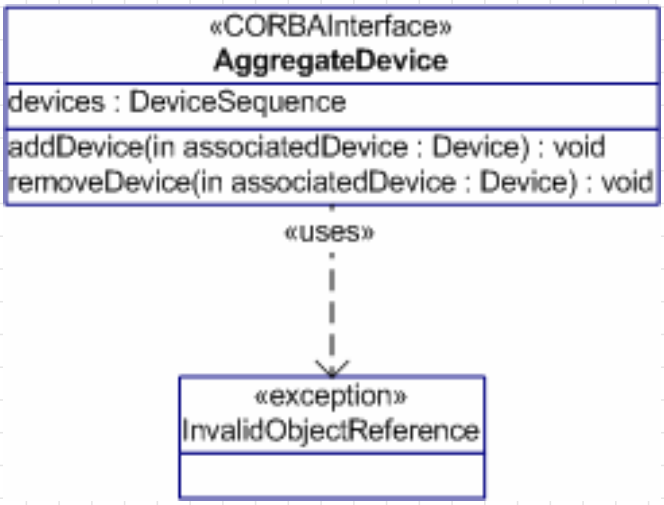
The *AggregateDevice* interface provides the required behavior that is needed to add and remove child devices from a parent device.

This interface may be provided via inheritance for any device that is used as a parent device.

Child devices use this interface to add or remove themselves to a parent device when being created or torn-down.

E.g., consider several FPGAs on a single card. Each FPGA would be a child device and the board that contains the FPGAs would be the parent device.

AggregateDevice UML



AggregateDevice Attribute and Operations

```
readonly attribute DeviceSequence devices;
```

It contain a list of devices that have been added to this device.

```
void addDevice (in Device associatedDevice) raises
(InvalidObjectReference);
```

The addDevice operation adds the input associatedDevice parameter to the *AggregateDevice*'s devices attribute. The associatedDevice is ignored when duplicated.

```
void removeDevice (in Device associatedDevice) raises
(InvalidObjectReference);
```

The removeDevice operation removes the input associatedDevice parameter from the *AggregateDevice*'s devices attribute.

Framework Control Interfaces

We will proceed with the description of the Core Framework by discussing the Framework Control Interfaces:

- *Application*
- *ApplicationFactory*
- *DeviceManager*
- *DomainManager*

Domain & Node

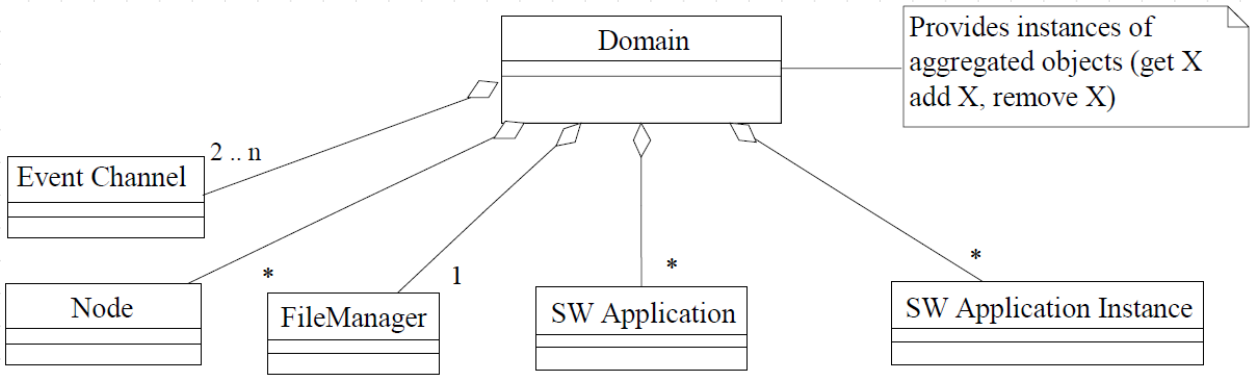
Domain

A Domain consists of software (SW) Applications (installed services), Nodes (and their devices, file systems, and services) File Manager, Event Channel, and SW Application Instances.

Node

A Node is an abstraction of the computing node, that allows the control and monitoring of the node resources (CPU, processes, memory and file systems), implements the OE, executes a DeviceManager and installed OE & CF services (CORBA Naming Service, FileSystem, Log, Event Service,...).

Domain



Services of a Domain

The services provided by a Domain are:

1. Installing and uninstalling application software onto the domain's file manager.
2. Retrieving Nodes (*DeviceManager*), SW Applications, and SW Application Instances.
3. Creating, Terminating, and Controlling SW Application Instances.
4. Registering and unregistering Nodes (*DeviceManager*) along with their Devices and services.
5. Registering and unregistering to the event channels. In the Domain there are two event channels by default: Outgoing Domain Event Channel and Incoming Domain Event Channel.

Mapping of Domain Services

The Domain services are mapped to the following CF interfaces:

1. *DomainManager* provides the services for retrieving, installing/registering, and uninstalling/unregistering Domain elements.
2. *Application* provides the services for terminating and controlling the SW Application Instance.
3. *ApplicationFactory* provides services for the creation of an Application. For each SW Application that is installed in the Domain, an ApplicationFactory object is created that is used for deploying the application within the Domain.
4. *DeviceManager* provides the services for managing a node.
5. *Device* interfaces (Device, LoadableDevice, ExecutableDevice, AggregateDevice) provides the services for managing hardware devices.

Framework Control Interfaces: *Application*

The *Application* interface provides the necessary operations for managing application lifecycle, state, and behavior of waveforms and services, that are the primary functional elements of a JTRS radio from an end-user perspective.

An *Application* has characteristics very similar to a *Resource*.

In general, the implementation of an *Application* is a set of *Resource* component implementations and their interconnections described by the SAD profile.

The *Application* behaves as the proxy for the instantiated software assembly.

Application developers don't have to develop code to tear down their application and to behave according to the Application's software profile (SAD).

Framework Control Interfaces: *Application*

The *Application* delegates its *Resource* operations to a *Resource* component that has been identified as the Assembly Controller for the software assembly.

The design and implementation of the assembly is totally under the control of an application developer without the need to worry about deployment management behavior.

The *Application*'s ports provide the capability of connecting the *Application* up to other components such as an *Application*.
The ports can also be used to provide additional command/control and/or status interfaces for an *Application*.

The *Application* sends out notification by the Outgoing Domain event channel of when an *Application* is destroyed. This allows for clients (HCI) to become immediately aware of an *Application* no longer available.

Framework Control Interfaces: *Application*

The *Application* interface provides the interface for the control, configuration, and status of an instantiated application in the domain.

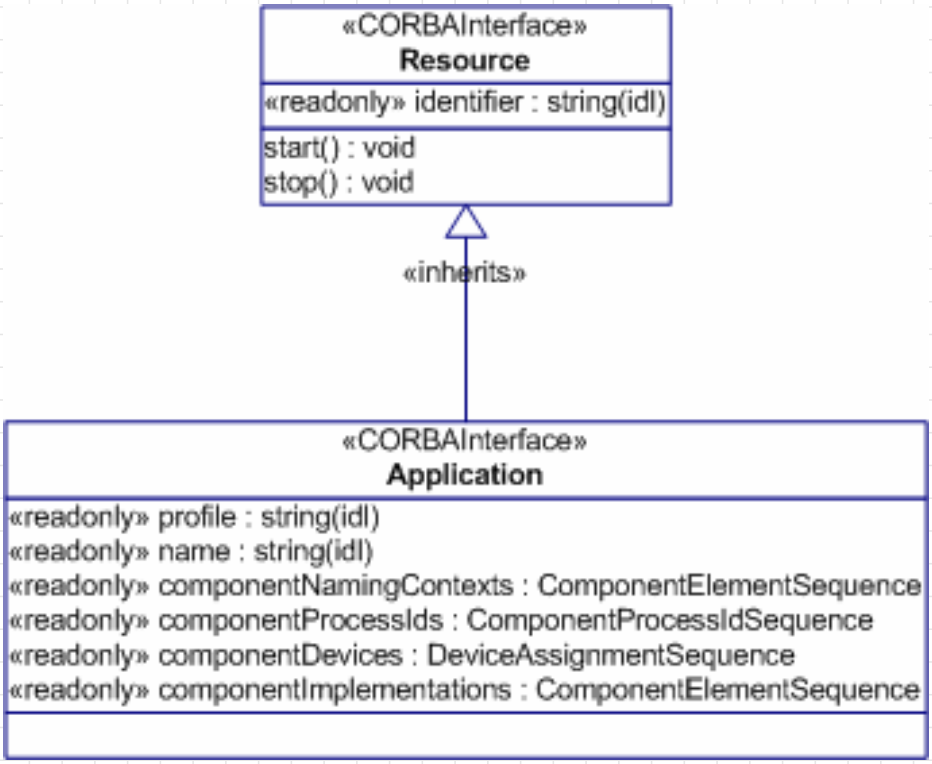
The *Application* interface inherits the IDL interface of *Resource*. A created application instance may contain *Resource* components and/or non-CORBA components.

The *Application* interface extended the *Resource* interface by adding deployment information (components associated with what devices, naming context names for components, etc.) of how the software assembly got deployed.

The *Application* interface `releaseObject` operation provides the interface to release the computing resources and devices allocated during the instantiation of the application.

An instance of an *Application* is returned by the `create` operation of an instance of the *ApplicationFactory* class.

Application UML



Application Attributes

```
readonly attribute string profile;
```

The readonly `profile` attribute shall contain a profile element (Profile Descriptor) with a file reference to the application's SAD file.

```
readonly attribute string name;
```

This readonly `name` attribute shall contain the name of the created application. The *ApplicationFactory* interface's `create` operation `name` parameter provides the name content.

Application Attributes

`readonly attribute ComponentElementSequence componentNamingContexts;`

The componentNamingContexts attribute shall contain the list of components' Naming Service Context within the application for those components using CORBA Naming Service.

```
struct ComponentElementType
  {
    string componentId;
    string elementId;
  };

typedef sequence <ComponentElementType> ComponentElementSequence;
```


Application Attributes

`readonly attribute ComponentProcessIdSequence componentProcessIds;`

The componentProcessIds attribute shall contain the list of components' process IDs within the Application for components that are executing on a device.

```
struct ComponentProcessIdType
  {
    string componentId;
    unsigned long processId;
  };

typedef sequence <ComponentProcessIdType> ComponentProcessIdSequence;
```

Application Attributes

`readonly attribute DeviceAssignmentSequence componentDevices;`

The componentDevices attribute shall contain a list of devices, which each component either uses, is loaded on or is executed on.

```
  struct DeviceAssignmentType
    {
      string componentId;
      string assignedDeviceId; //device where the component is loaded/executed
    };

typedef sequence <DeviceAssignmentType> DeviceAssignmentSequence;
```


Application Attributes

`readonly attribute ComponentElementSequence componentImplementations;`

The componentImplementations attribute shall contain the list of components' SPD implementation IDs within the application for those components created.

Application Operations

The operations of the Application interface are those inherited from the *Resource* interface.

The application shall delegate the implementation of the inherited *Resource* operations (`runTest`, `start`, `stop`, `configure`, and `query`) to the Application Resource component identified by the application's SAD assemblycontroller element (Assembly Controller).

The application shall propagate exceptions raised by the application's Assembly Controller's operations.

The `initialize` operation shall not be propagated to the application's components or its Assembly Controller.

Application Operations

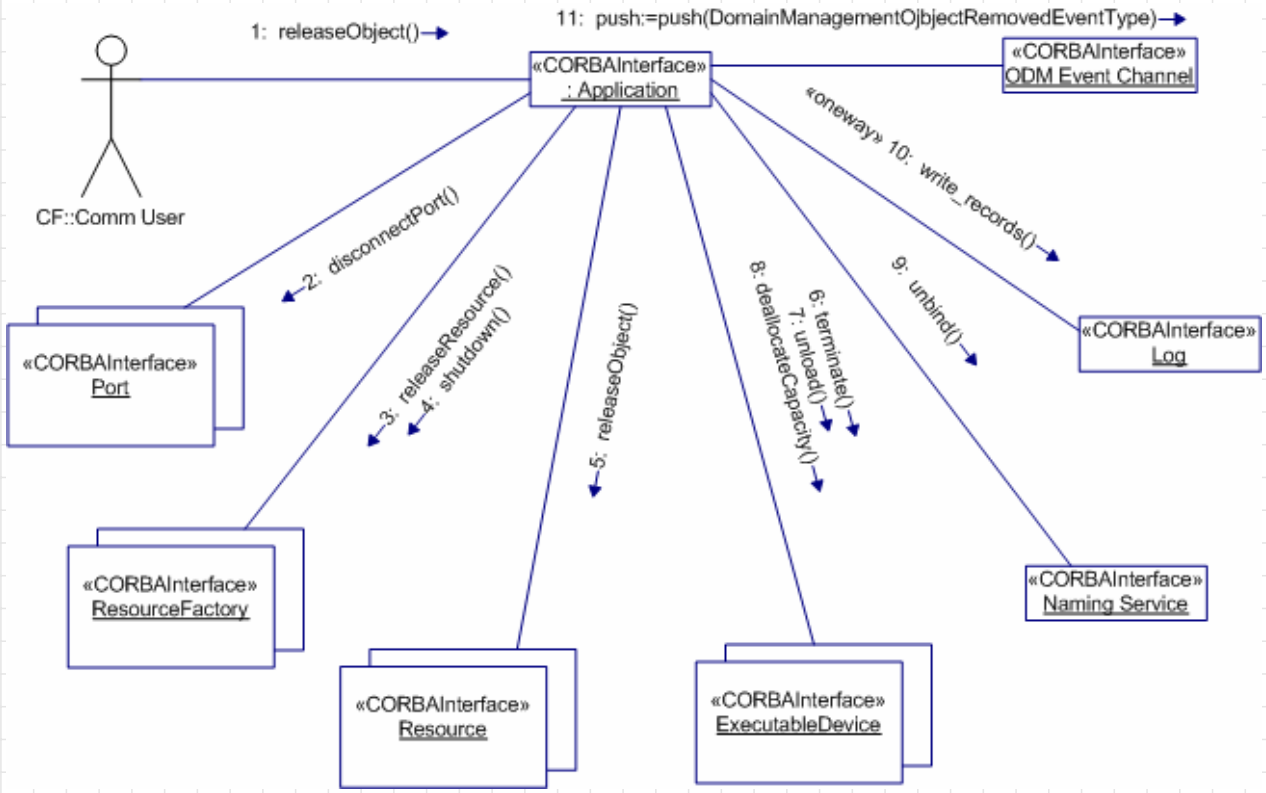
```
void releaseObject() raises (ReleaseError);
```

The `releaseObject` operation:

- terminates execution of the application,
- returns all allocated computing resources,
- de-allocates the resources' capacities in use by the devices associated with the application,
- removes the message connectivity with its associated applications (e.g., ports, resources, and logs) in the domain.

The above behavior is in addition to the `LifeCycle::releaseObject` operation behavior.

Application Operations: releaseObject



Application Operations: getPort

```
Object getPort (in string name) raises (UnknownPort);
```

The getPort operation obtains an object reference to a specific visible port of the application.

The getPort operation shall return object references only for input port names that match the port names that are in the application SAD *externalports* element.

Framework Control Interfaces: *ApplicationFactory*

The *Application* interface cannot be responsible for the actual creation of an application, since an executable instance doesn't exist yet.

Users must be aware of the types of applications available in the radio and then be able to command the creation of a new instance of a selected application type.

The *ApplicationFactory* provides an interface to request the creation of a specific type of application in the domain.
The user (through the user interface) can create an application instance, provide it with pre-selected devices, and specify its configuration properties.

Framework Control Interfaces: *ApplicationFactory*

The *ApplicationFactory* interface was designed based on the Factory Design Pattern.

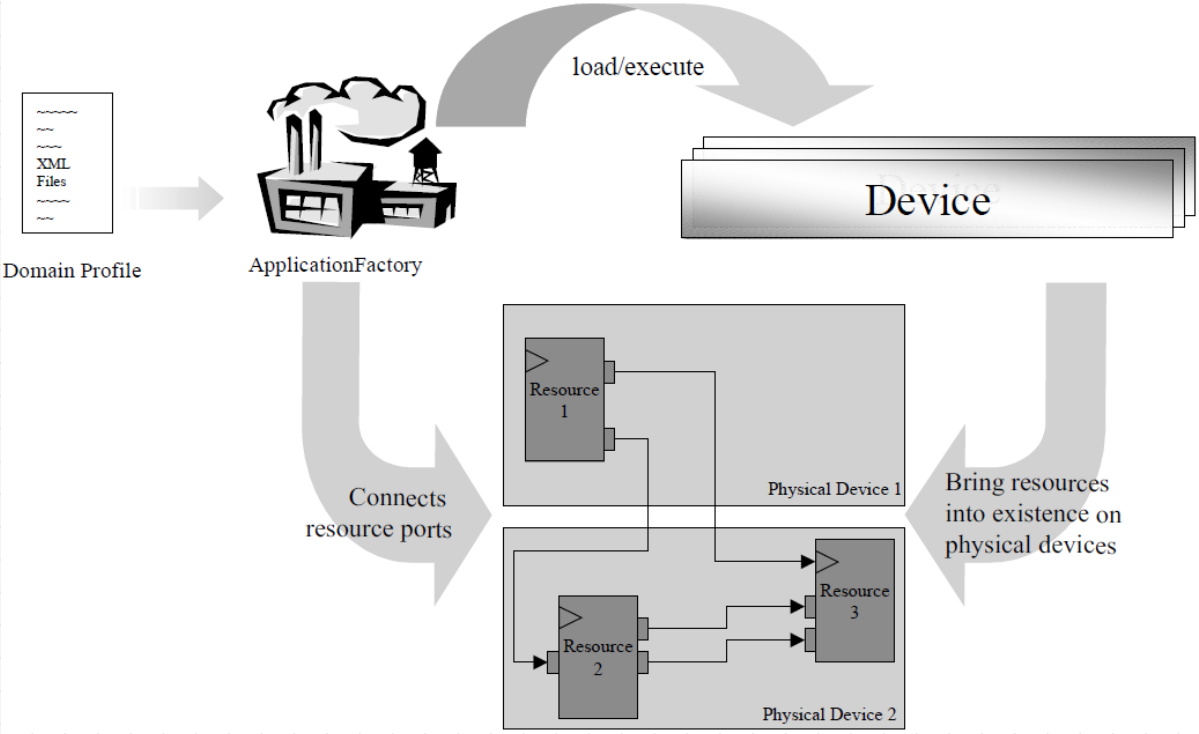
There is one *ApplicationFactory* object for each type of application.

The *ApplicationFactory* creates a CORBA object implementing the Application interface for each application created.

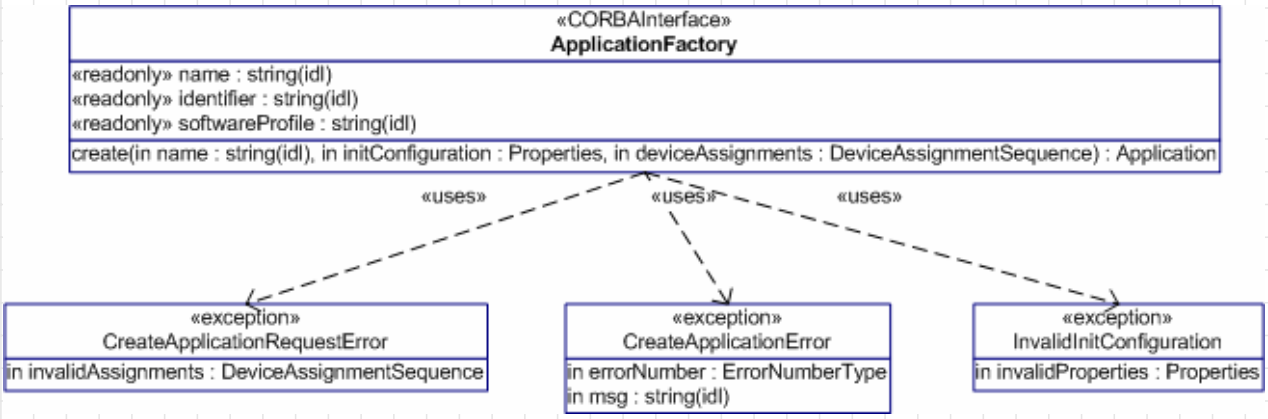
Application developers do not have to develop code to parse their own software profiles to create their application within a domain. An *ApplicationFactory* CORBA object is created for each different Software Assembly Descriptor (SAD) XML file installed in the domain.

The *ApplicationFactory* forms Naming Contexts, which application's components use to place their CORBA object references. This provides the capability of instantiating an application multiple times. Each application' component instantiation uses the same code but a different naming context CORBA object in which to place their object references.

Framework Control Interfaces: *ApplicationFactory*



ApplicationFactory UML



Note that *ApplicationFactory* is not derived from *Resource*.

ApplicationFactory Attributes

```
readonly attribute string name;
```

The readonly name attribute contains the user-friendly name of the application instantiated by an application factory. The name attribute shall be identical to the *softwareassembly* element name attribute of the application's Software Assembly Descriptor file.

```
readonly attribute string softwareProfile;
```

The readonly softwareProfile attribute shall contain a profile element (Profile Descriptor) with a file reference to the application's SAD file.

```
readonly attribute string identifier;
```

The readonly identifier attribute shall contain the unique identifier for an ApplicationFactory instance. The identifier shall be identical to the *softwareassembly* element id attribute of the application factory's SAD file.

ApplicationFactory Operations: `create`

```
Application create (in string name, in Properties
initConfiguration, in DeviceAssignmentSequence
deviceAssignments) raises (CreateApplicationError,
CreateApplicationRequestError, InvalidInitConfiguration);
```

The create operation is used to create an application within the system domain.

The create operation provides a client interface to request the creation of an application on client requested device(s) and/or the creation of an application in which the application factory determines the necessary device(s) required for instantiation of the application.

It returns an *Application* reference for the created application.

ApplicationFactory Operations: `create`

The precise behavior of the `create` operation is quite complex.

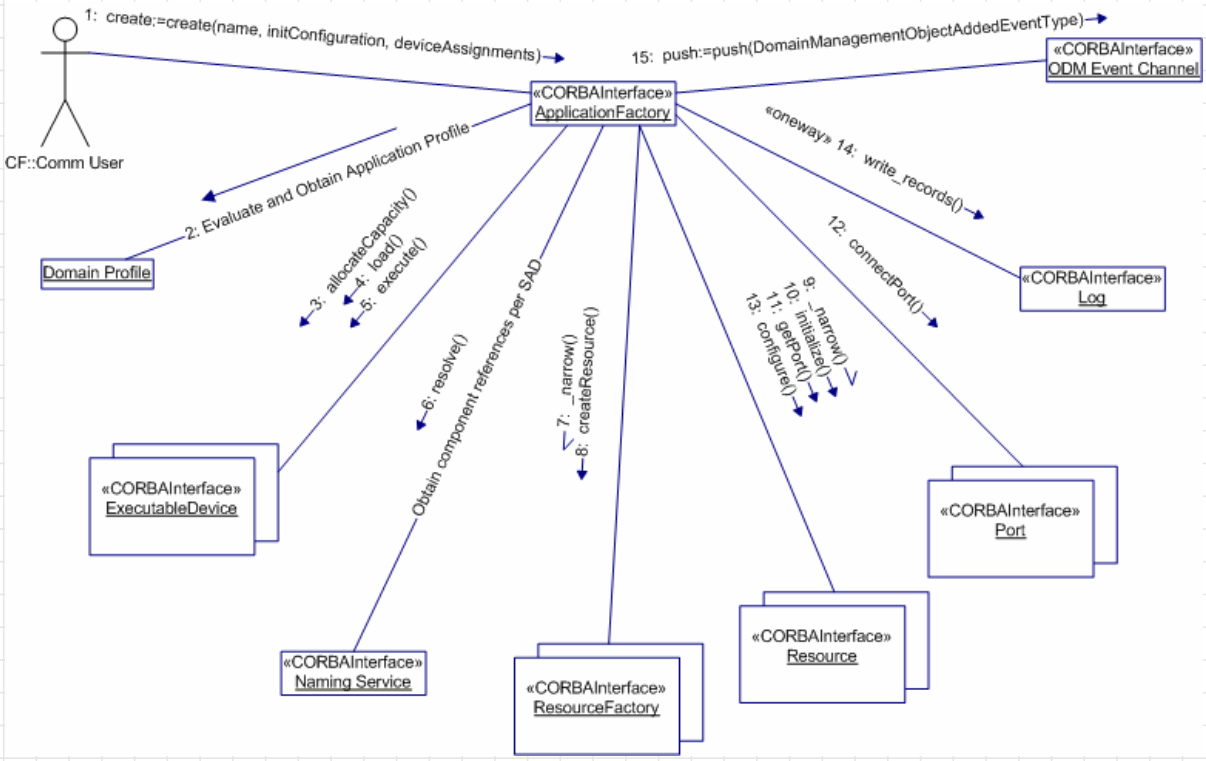
The following steps demonstrate one scenario of the behavior of an application factory for the creation of an application:

1. Client invokes the create operation.
2. Evaluate the Domain Profile for available devices that meet the application's memory and processor requirements, available dependent applications and libraries needed by the application. Create an instance of an *Application*, if possible. Update the memory and processor utilization of the devices.
3. Allocate the device(s) memory and processor utilization.
4. Load the application software modules on the devices using the appropriate Device(s) interface provided the application software modules haven't already been loaded.
5. Execute the application software modules on the devices using the appropriate Device interface.

ApplicationFactory Operations: `create`

7. Obtain the object reference (*Resource* or *ResourceFactory*) as described by the SAD.
8. If the component obtained from the CORBA Naming Service is a resource factory, narrow it to be a *ResourceFactory*.
9. If the component is a *ResourceFactory*, then create a resource using the *ResourceFactory* interface.
10. If the components obtained from the Naming Services is a *Resource*, narrow it to be a *Resource*.
11. Initialize the resource.
12. Get Port object references for the resources.
13. Connect the ports that interconnect the resources' ports together.
14. Configure the assemblycontroller component.
15. Write a log message on successful application creation.
16. Generate an event to indicate the application has been added to the domain.
17. Return the Application object reference.

ApplicationFactory Operations: `create`



Framework Control Interfaces: *DeviceManager*

The *DeviceManager* is a service that manages a set of persistent logical Devices and services (e.g., Log Service, Event Service, Naming Service, etc.) for a given node within a system or domain.

A given system is composed of a set of nodes. These nodes are associated with a *DeviceManager*.

The *DeviceManager* provides the capability of simultaneously starting up logical Devices and services on a node at power up.

As nodes are removed or added to the system (*DomainManager*), the set of elements belonging to a node are easily identified by the attributes of the *DeviceManager* interface.

Framework Control Interfaces: *DeviceManager*

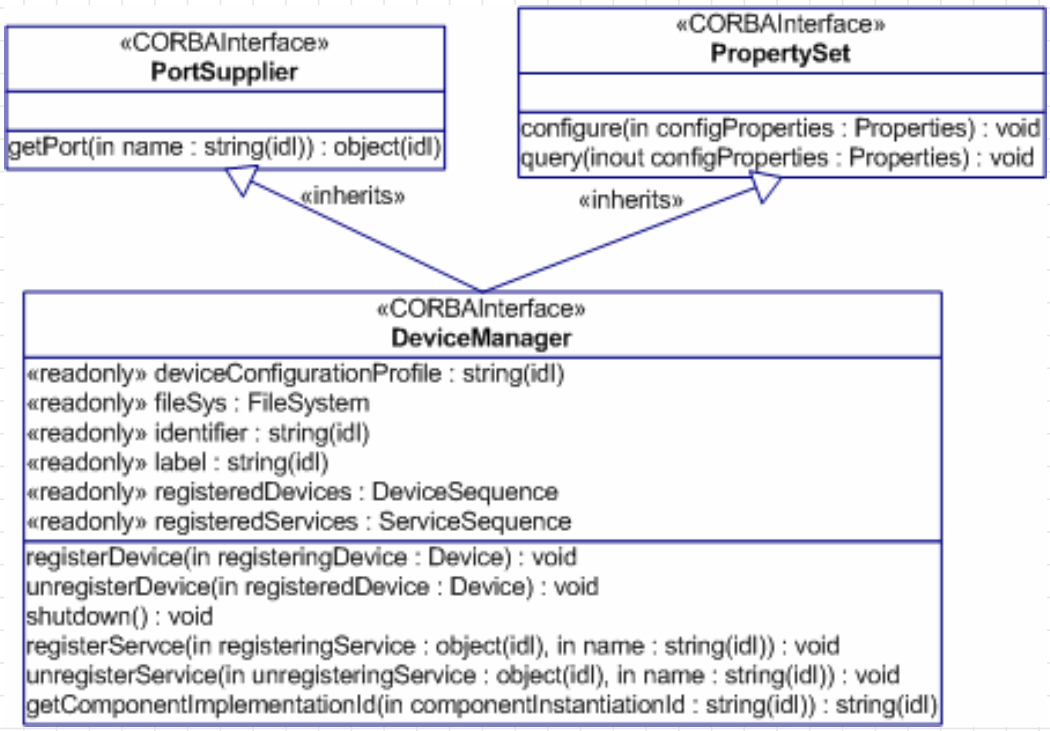
The *DeviceManager* provides the capability of changing the characteristic of the node by its associated Device Configuration Descriptor (DCD) XML file and by its operations (services and logical devices can be added or removed).

A node usually has some file system associated with it. Therefore the *DeviceManager* interface has a file system attribute.

The *DeviceManager* interface inherits the *PropertySet* interface in order to manage implementation properties that are described in its Software Package Descriptor (SPD) file.

The *PortSupplier* interface inherited by the DeviceManager interface is used to connect services (e.g., Log) to the *DeviceManager*.

DeviceManager UML



DeviceManager General Behaviour

The *Device manager* upon start up shall register itself with a Domain Manager.

This requirement allows the system to be developed where at a minimum only the *DomainManager*'s object reference needs to be known.

A *Device Manager* shall use the information in the *Device Manager*'s DCD for determining:

- Services and Devices to be deployed end executed;
- Devices to be aggrated to another device;
- Mount point names for file systems;
- The *Device Manager*'s identifier and label attribute values.

DeviceManager General Behaviour

The *DeviceManager* shall create FileSystem components implementing the *FileSystem* interface for each OS file system, as stated in the DCD.

If multiple file systems, the *DeviceManager* shall mount them to a *FileManager* component (widened to a FileSystem through the FileSys attribute).

The *DeviceManager* shall launch each executable devices and services specified in the DCD.

Eventually, the *DeviceManager* shall register itself at the *DomainManager*.

DeviceManager General Behaviour

For each launched device, the *DeviceManager* supplies the execute operation parameters consisting of:

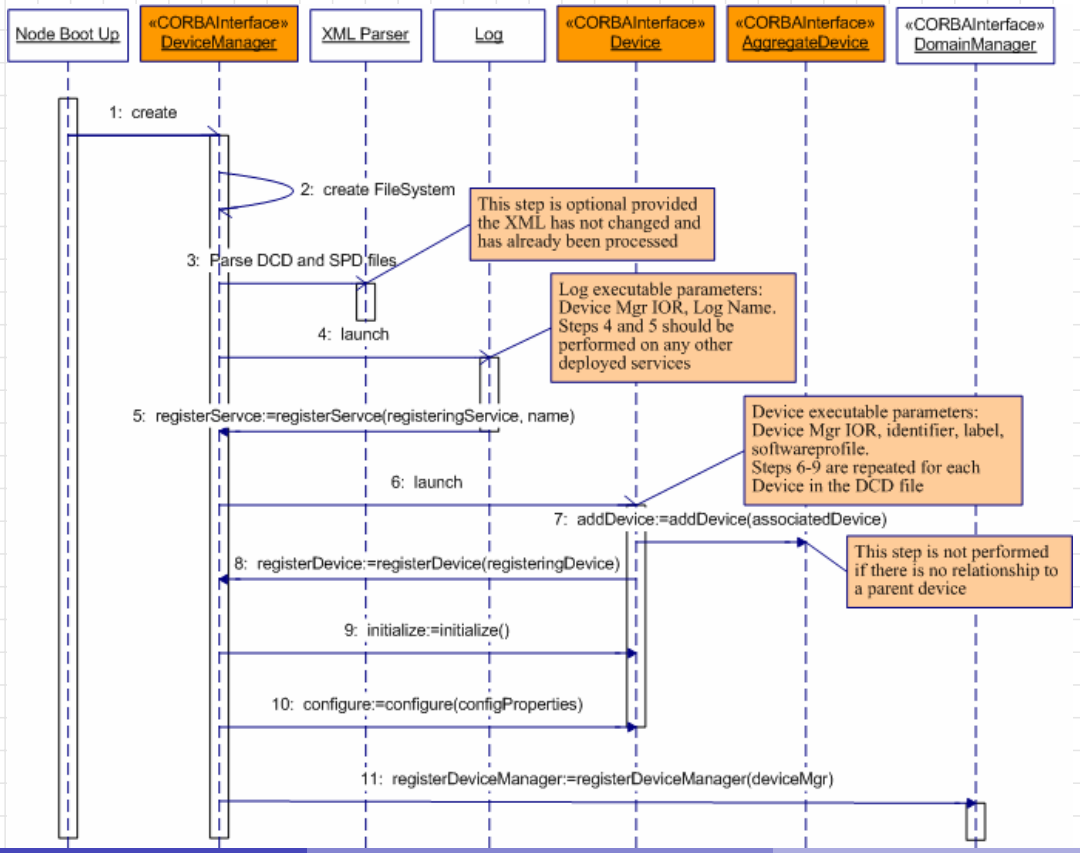
1. Device Manager IOR – The ID is "DEVICE_MGR_IOR" and the value is a string that is the *DeviceManager* stringified IOR.
2. Profile Name – The ID is "PROFILE_NAME" and the value is the full mounted file system file path name.
3. Device Identifier – The ID is "DEVICE_ID" and the value is a string whith the id attribute.
4. Device Label – The ID is "DEVICE_LABEL" and the value is a string.
5. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is an *AggregateDevice* stringified IOR. This parameter is only used for child devices.
6. The execute properties as specified in the DCD.

DeviceManager General Behaviour

For each launched service, the *DeviceManager* supplies the execute operation parameters consisting of:

1. Device Manager IOR – The ID is "DEVICE_MGR_IOR" and the value is a string that is the *DeviceManager* stringified IOR.
2. Service Name – The ID is "SERVICE_NAME" and the value is a string.
3. The execute properties as specified in the DCD.

DeviceManager Startup



DeviceManager Attributes

The interface for a *DeviceManager* is based upon its attributes, which are:

1. Device Configuration Profile - a mapping of physical device locations to meaningful labels (e.g., audio1, serial1, etc.), along with the devices and services to be deployed.
2. File System - the FileSystem associated with this device manager.
3. Device Manager Identifier - the instance-unique identifier for this device manager.
4. Device Manager Label - the meaningful name given to this device manager.
5. Registered Devices - a list of devices that have registered with this device manager.
6. Registered Services - a list of services that have registered with this device manager

DeviceManager Attributes

```
readonly attribute string identifier;

readonly attribute string label;

readonly attribute FileSystem fileSys;

readonly attribute string deviceConfigurationProfile;
```

The readonly `deviceConfigurationProfile` attribute shall contain a *profile* element (Profile Descriptor) with a file reference to the device manager's Device Configuration Descriptor (DCD) file.

DeviceManager Attributes

```
readonly attribute DeviceSequence registeredDevices;
```

The readonly registeredDevices attribute shall contain a list of devices that have registered with this device manager or a sequence length of zero if no devices have registered with the device manager.

```
readonly attribute ServiceSequence registeredServices;
```

The readonly registeredServices attribute shall contain a list of services that have registered with this device manager or a sequence length of zero if no services have registered with the device manager.

```
struct ServiceType
  {
    Object serviceObject;
    string serviceName;
  };


typedef sequence <ServiceType> ServiceSequence;
```

DeviceManager Operations: registerDevice, unregisterDevice

```
void registerDevice (in Device registeringDevice) raises
(InvalidObjectReference);
```

The registerDevice operation shall add the input registeringDevice to the registeredDevices attribute if it is not already present.

It shall register the registeringDevice with the Domain Manager when the device manager has already registered to the domain manager.

```
void unregisterDevice (in Device registeredDevice) raises
(InvalidObjectReference);
```

The unregisterDevice operation shall remove the input registeredDevice from the registeredDevices attribute and from the Domain Manager when the input registeredDevice is registered with the Device Manager.

DeviceManager Operations: registerService, unregisterService

```
void registerService (in Object registeringService, in
string name) raises (InvalidObjectReference);
```

The registerService operation shall add the input registeringService to the registeredServices attribute if it is not already present.

It shall register the registeringService with the Domain Manager when the Device Manager has already registered to the domain manager.

```
void unregisterService (in Object unregisteringService, in
string name) raises (InvalidObjectReference);
```

The unregisterService operation shall remove the input registered service specified by the input name parameter from the registeredServices attribute and from the Domain Manager.

DeviceManager Operations: `shutdown`

```
void shutdown();
```

The `shutdown` operation provides the mechanism to terminate a device manager.

The `shutdown` shutdown operation shall unregister the Device Manager from the Domain Manager.

It perform `releaseObject` on all of the registered devices in the *D*eviceManager's `registeredDevices` attribute.

The `shutdown` operation shall cause the Device Manager to be unavailable (i.e. released from the CORBA environment and its process terminated on the OS), when all of the device manager's registered devices.

DeviceManager Operations: `getComponentImplementationID`

```
string getComponentImplementationId (in string
componentInstantiationId);
```

The `getComponentImplementationId` operation returns the SPD *implementation* ID that the *DeviceManager* interface used to create a component identified by the input string `componentInstantiationId`.

Note that a Software Package may contain multiple implementations (for different hardware) using the same properties and Sofware Component Descriptor.

DeviceManager Operations: getComponentImplementationID

Here an example from a SPD file:

```
...
 <descriptor name="">
       <localfile name="BasicTestDevice.scd.xml" />
   </descriptor>

 <implementation id="DCE:f7cd1dd5-ea37-4a56-9e68-bedd89acdff9"
          aepcompliance= "aep_compliant">
       <code type="Executable">
           ...
       </code>
       <compiler name="gcc" version="3.4" />
       ...
   </implementation>

 <implementation id="DCE:0ef71fab-731d-4ee1-a528-a6da2207e0c5"
          aepcompliance="aep_compliant">
       ...

   </implementation>
...
```


Framework Control Interfaces: *DomainManager*

The *DomainManager* provides the repository for the elements within the system (or domain).

These elements are the installed application, application instances, DeviceManagers (nodes and their devices and services) and event channels.

The *DomainManager* provides operations for the elements to register themselves. The registration technique optimizes the operation of the DomainManager because it does have to expend processor resources polling for new elements.

When elements registered to the DomainManager, the DomainManager uses the elements' XML profiles (SAD, SPD, DCD) to obtain their deployment characteristics.

Framework Control Interfaces: *DomainManager*

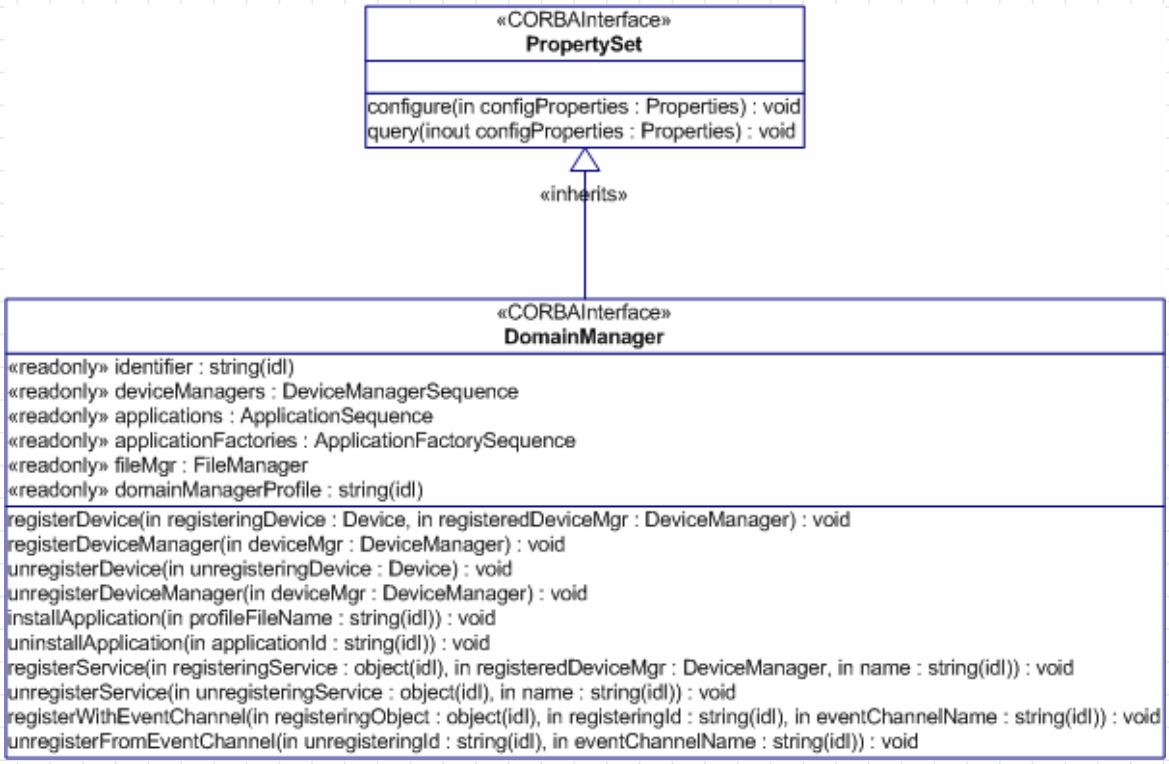
As *DeviceManagers*, *Devices*, and services registered to the *DomainManager*, the *DomainManager* establishes connections to services for these elements instead of *DeviceManagers*. This technique provides the most efficient technique since the *DomainManager* knows when services become available.

Connections established for services are for the Log and event channels.

The *DomainManager* sets up the Incoming Domain and Outgoing Domain event channels. This allows for efficient implementations of the DomainManager of knowing when Devices' state changes. It also allows for asynchronous notification of system changes to the outside clients (HCI).

The *DomainManager* receives information from various elements in the architecture, including all the *DeviceManagers* (DCD Profiles, Device SPD Profiles), and the Install applications (SAD Profiles).

DeviceManager UML



DomainManager Attributes

```
readonly attribute string identifier;
```

```
readonly attribute FileManager fileMgr;
```

```
readonly attribute string domainManagerProfile;
```

The `domainManagerProfile` attribute contains a *profile* element (Profile Descriptor) with a file reference to the DomainManager Configuration Descriptor (DMD) file.

Files referenced within the profile are obtained via the domain manager's *FileManager*.

DomainManager Attributes

```
readonly attribute ApplicationSequence applications;
```

The applications attribute is read-only containing a sequence of instantiated Applications in the domain.

```
readonly attribute ApplicationFactorySequence
applicationFactories;
```

The readonly `applicationFactories` attribute shall contain a list with one application factory per application (SAD file and associated files) successfully installed.

```
readonly attribute DeviceManagerSequence deviceManagers;
```

The readonly `deviceManagers` attribute shall contain a list of registered device managers that have registered with the domain manager.

DomainManager Operations: registerDeviceManager

```
void registerDeviceManager (in DeviceManager deviceMgr)
raises (InvalidObjectReference, InvalidProfile,
RegisterError );
```

The registerDeviceManager operation is used to register a device manager, its device(s), and its services.

The registerDeviceManager operation shall:

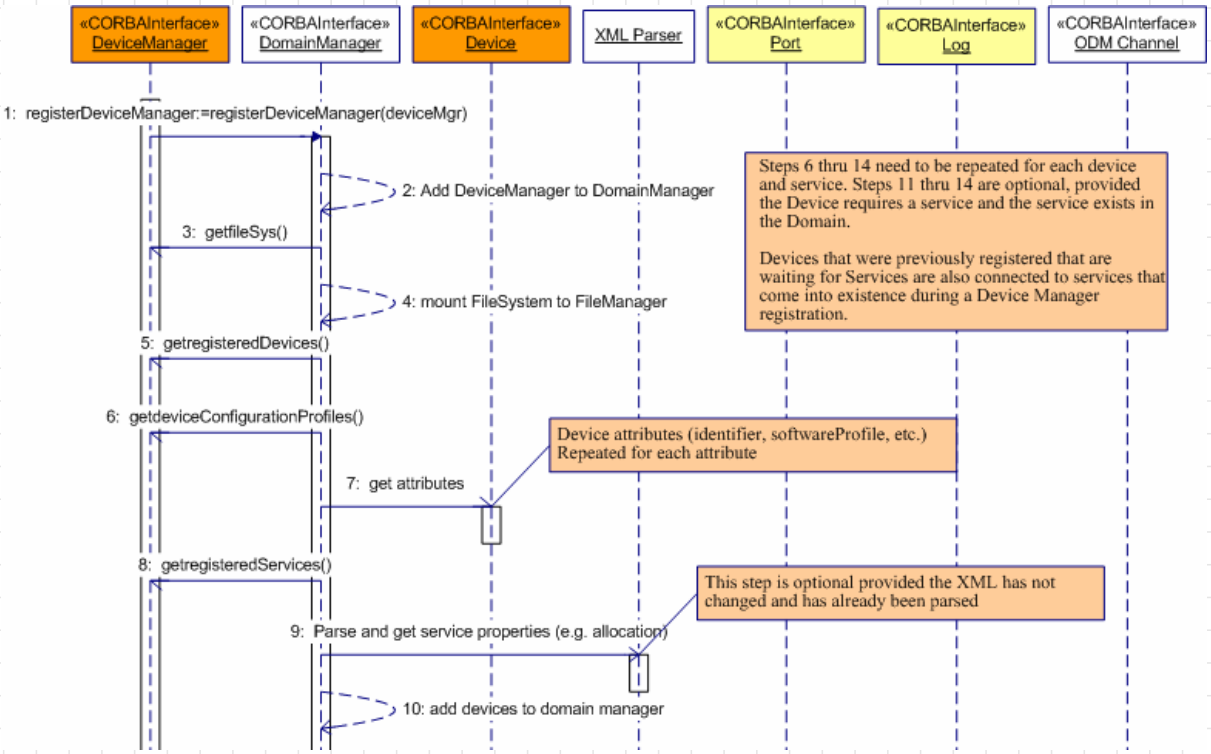
1. add the device manager (deviceMgr parameter) to the deviceManagers attribute, if it does not already exist.

   The registerDeviceManager operation shall add the input device manager's registered devices and services to the domain manager. The domain manager associates the devices and service with the device manager in order to supportunregisterDeviceManager.
2. return without exception and do nothing when that device manager has the same identifier as a previously registered device manager.

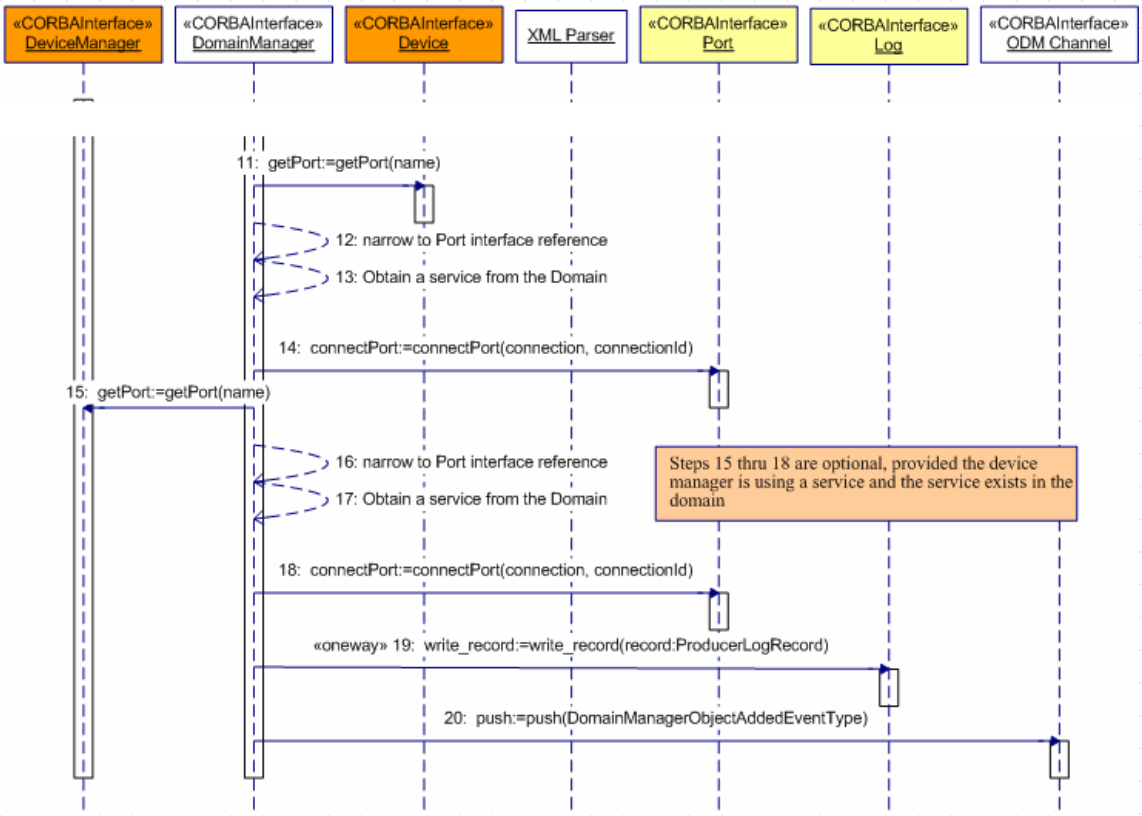
DomainManager Operations: registerDeviceManager

3. establish any connections for the device manager indicated by the input deviceMgr parameter, which are specified in the connections element of the device manager's DCD file, that are possible with the current set of registered devices and services.
4. obtain all the software profiles from the registering device manager's file systems.
5. mount the device manager's file system to the domain manager's file manager. The mounted FileSystem name shall have the format, "/DomainName/HostName", where DomainName is the name of the domain and HostName is the input deviceMgr's label attribute.
6. write a FAILURE_ALARM log record to a domain manager's Log, upon unsuccessful device manager registration.
7. send an event to the Outgoing Domain Management event channel upon successful registration of a device manager.

registerDiviceManager UML



registerDiviceManager UML



DomainManager Operations: registerDevice

```
void registerDevice (in Device registeringDevice, in
DeviceManager registeredDeviceMgr) raises
(InvalidObjectReference, InvalidProfile,
DeviceManagerNotRegistered, RegisterError);
```

The registerDevice operation is used to register a device for a specific device manager.

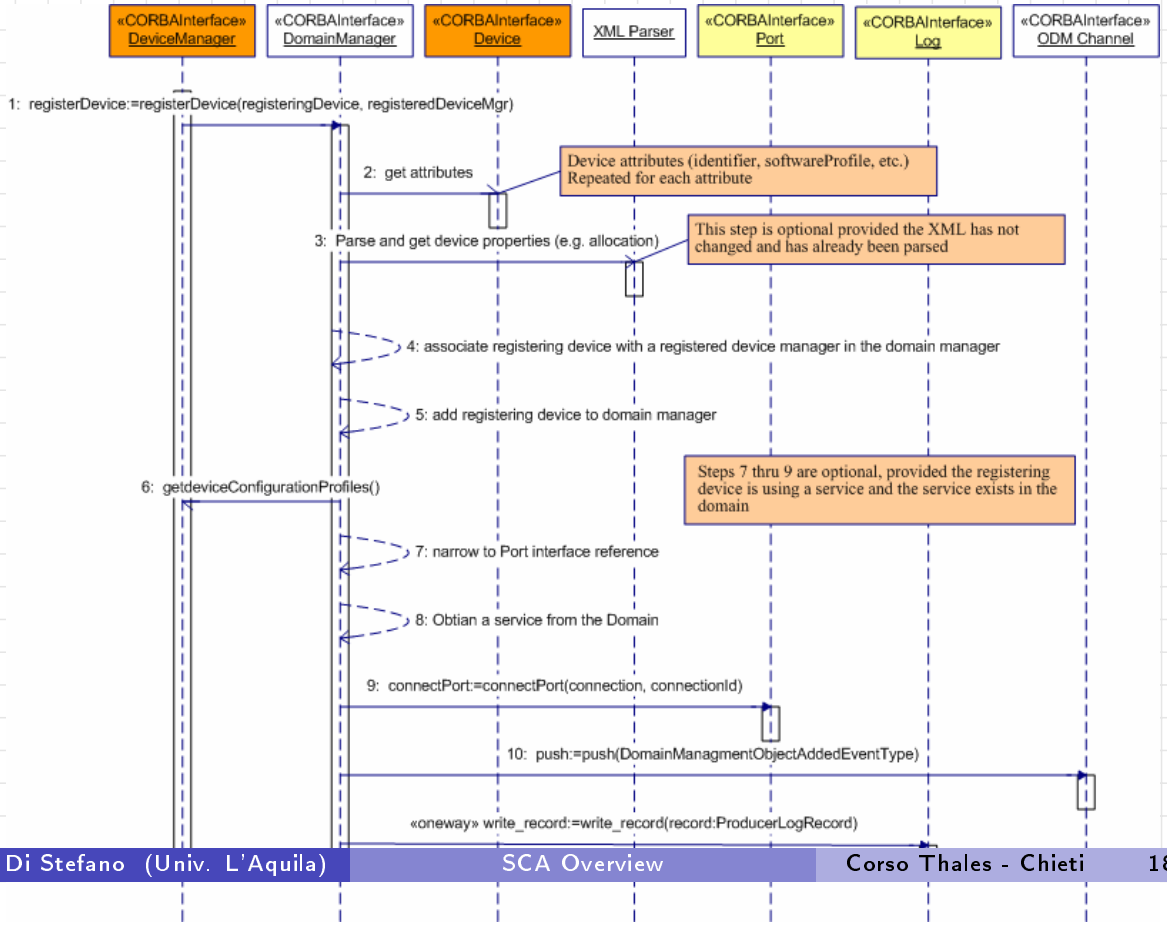
The registerDevice operation shall:

1. The registerDevice operation shall add the device indicated by the input registeringDevice parameter and the device's attributes to the domain manager, if it does not already exist.
2. The registerDevice operation associates the device indicated by the input registeringDevice parameter with the device manager indicated by the input registeredDeviceMgr parameter.

DomainManager Operations: registerDevice

3. The registerDevice operation shall establish any pending connections from previously registered device managers when the registering device completes these connections.
4. The registerDevice operation shall write an ADMINISTRATIVE_EVENT log record to a domain manager log upon successful device registration.
5. The registerDevice operation shall send an event to the Outgoing Domain Management event channel, upon successful registration of a device.

registerDivice UML

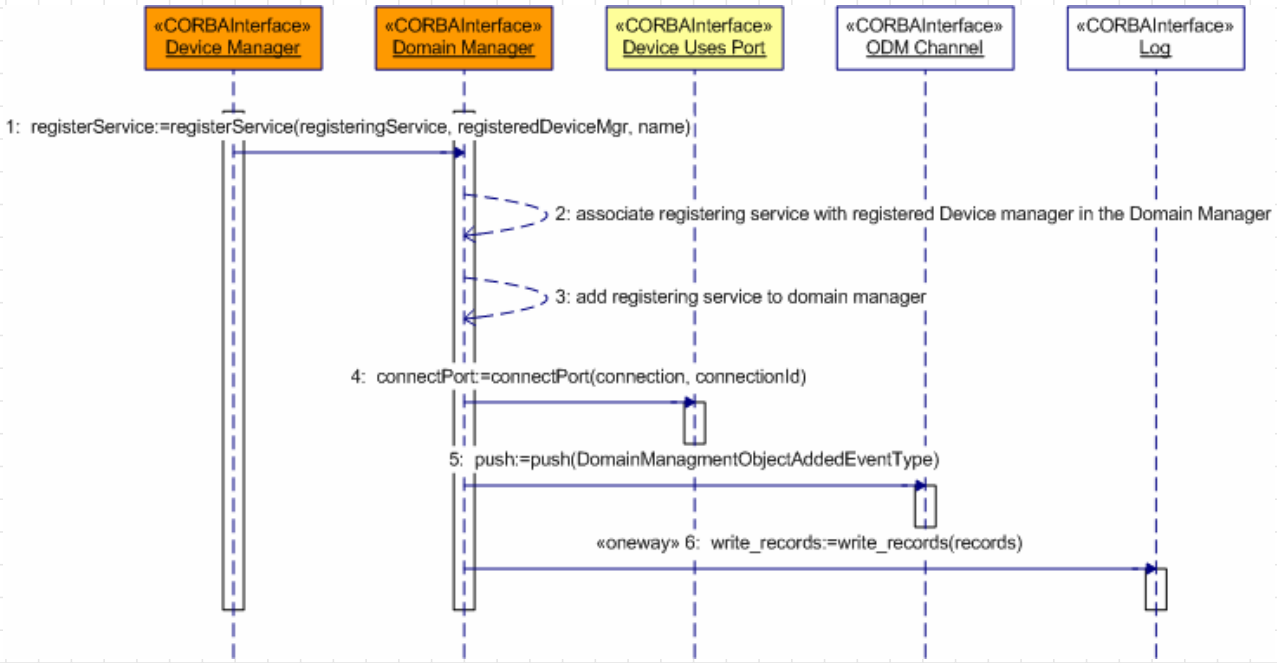


DomainManager Operations: `registerService`

```
void registerService (in Object registeringService, in
DeviceManager registeredDeviceMgr, in string name) raises
(InvalidObjectReference, DeviceManagerNotRegistered,
RegisterError);
```

Similar behavior as `registerDevice`, but for a service (see next slide).

registerService UML



DomainManager Operations: `installApplication`

```
void installApplication (in string profileFileName) raises
(InvalidProfile, InvalidFileName,
ApplicationInstallationError, ApplicationAlreadyInstalled);
```

The `installApplication` operation is used to install new application software (new *ApplicationFactory*) in the domain.

The input `profileFileName` parameter is the absolute pathname of the application SAD. It shall verify the existence of the application's SAD file and other files cited in it.

The `installApplication` operation shall send an event to the ODM event channel. For this event,

- The `producerId` is the identifier of the domain manager.
- The `sourceId` is the identifier of the installed application factory.
- The `sourceName` is the name of the installed application factory.
- The `sourceIOR` is the object reference for the installed application factory.
- The `sourceCategory` is "APPLICATION_FACTORY".

DomainManager unregister/unistall Operations

```
void unregisterDeviceManager (in DeviceManager deviceMgr)
raises (InvalidObjectReference, UnregisterError);

void unregisterDevice (in Device unregisteringDevice) raises
(InvalidObjectReference, UnregisterError);

void unregisterService (in Object unregisteringService, in
string name) raises (InvalidObjectReference,
UnregisterError);

void uninstallApplication (in string applicationId)raises
(InvalidIdentifier, ApplicationUninstallationError);
```


DomainManager Event Channel Operations

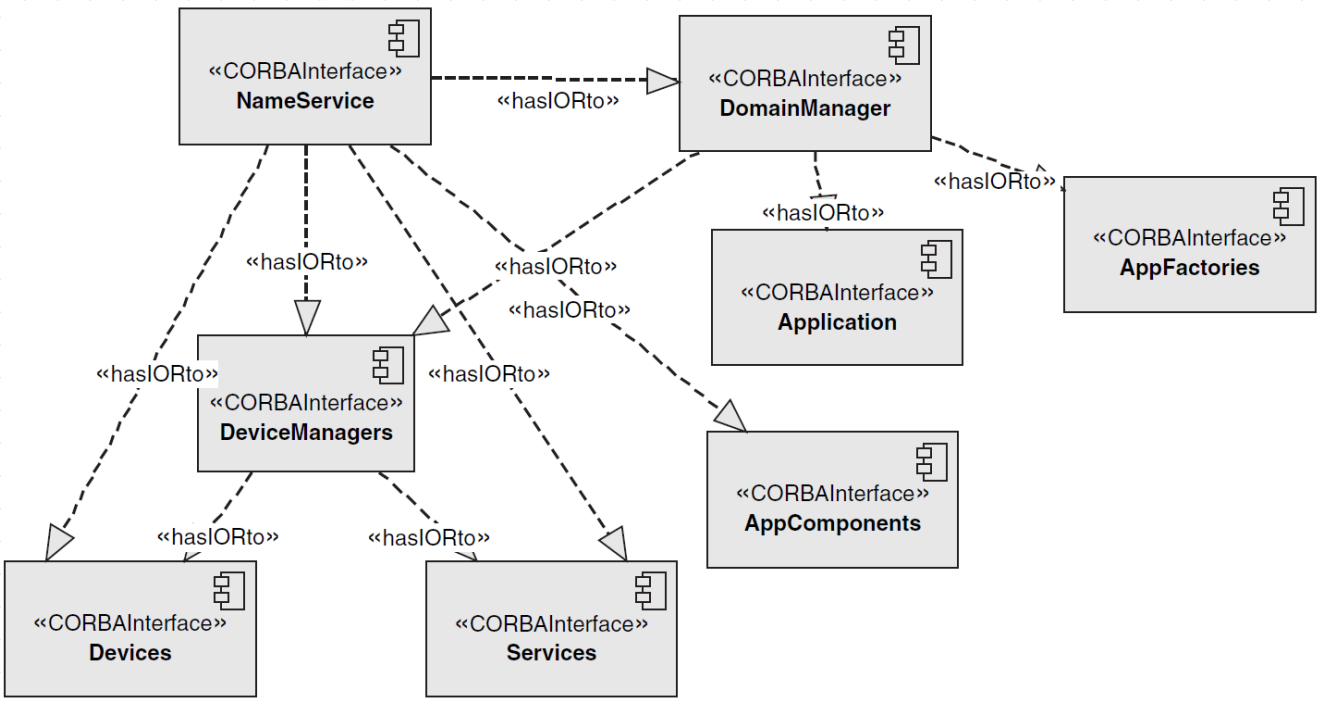
```
void registerWithEventChannel (in Object registeringObject,
in string registeringId, in string eventChannelName) raises
(InvalidObjectReference, InvalidEventChannelName,
AlreadyConnected);
```

The `registerWithEventChannel` operation is used to connect a consumer (`registeringObject`) to a domain's event channel (`eventChannelName`).

```
void unregisterFromEventChannel (in string unregisteringId,
in string eventChannelName) raises (InvalidEventChannelName,
NotConnected);
```

The `unregisterFromEventChannel` operation is used to disconnect a consumer from a domain's event channel.

DomainManager and IORs management



Framework Services Interfaces: *File*

The *File* interface provides the ability to read and write files residing within a compliant, distributed file system.

A file can be thought of conceptually as a sequence of octets with a current filePointer describing where the next read or write will occur. This filePointer points to the beginning of the file upon construction of the file object.

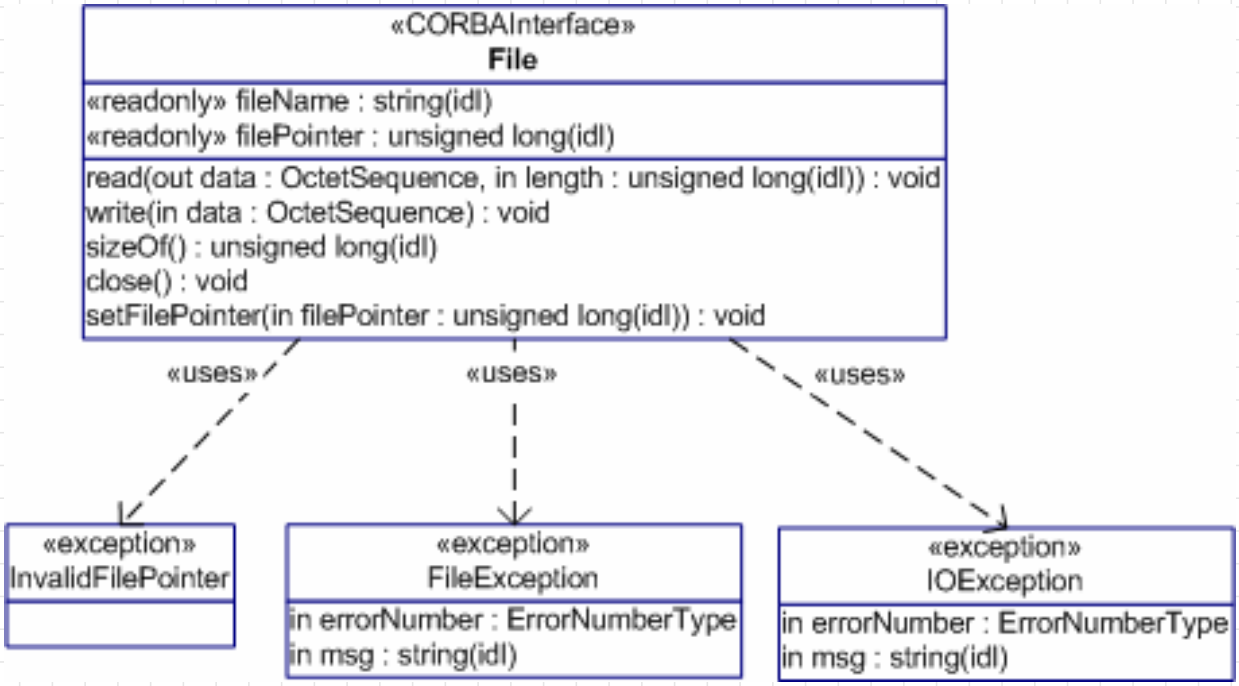
The File interface is modeled after the POSIX/C file interface.

The *File* interface is used when accessing CF elements' profile attributes, loading and executing files, installing applications, and for application's components usage.

The *File* interface abstracts away where the file object resides within the system.

Applications must use the CF File interfaces so that the location of the files is transparent to the application.

Framework Services Interfaces: *File*



File Attributes

```
readonly attribute string fileName;
```

The readonly `fileName` attribute contains the pathname used as the input `fileName` parameter of the `FileSystem::create` operation when the file was created .

```
readonly attribute unsigned long filePointer;
```

The readonly `filePointer` attribute contains the current file position, that is where the next read or write will occur.

File Operations

```
void read (out OctetSequence data, in unsigned long length)
raises (IOException);
```

The `read` operation reads the number of octets specified by the input length parameter and advance the value of the filePointer attribute by the number of octets actually read.

The operation shall read less than the number of octets specified in the input-length parameter, when an end of file is encountered.

The `read` operation shall return via the out `data` parameter a CF OctetSequence that equals the number of octets actually read from the File.

If the `filePointer` attribute reflects the end of the File, the read operation shall return a zero-length CF OctetSequence.

File Operations

```
void write (in OctetSequence data) raises (IOException);

unsigned long sizeOf() raises (FileException);

void setFilePointer (in unsigned long filePointer) raises
(InvalidFilePointer, FileException);

void close() raises (FileException);
```

The `close` operation shall release any OE file resources associated with the component.

The close operation shall make the file unavailable to the component.

Framework Services Interfaces: *FileSystem*

The *FileSystem* interface defines operations that enable *remote* access to a physical file system.

The files stored on a file system may be plain files or directories.

Valid individual filenames and directory names shall be 40 characters or less. Valid characters for a filename or directory name are the 62 alphanumeric characters (Upper, and lowercase letters and the numbers 0 to 9) in addition to "." , "_" and "-" characters. The filenames "." and ".." are invalid in the context of a file system.

Valid pathnames are structured according to the POSIX specification whose valid characters include the "/" (forward slash) character in addition to the valid filename characters. A valid pathname may consist of a single filename. A valid pathname shall not exceed 1024 characters.

Framework Services Interfaces: *FileSystem*

The *FileSystem* interface provides a distributed (network) file system service capability that is used when accessing CF elements' profile attributes, loading and executing files, and installing and uninstalling applications.

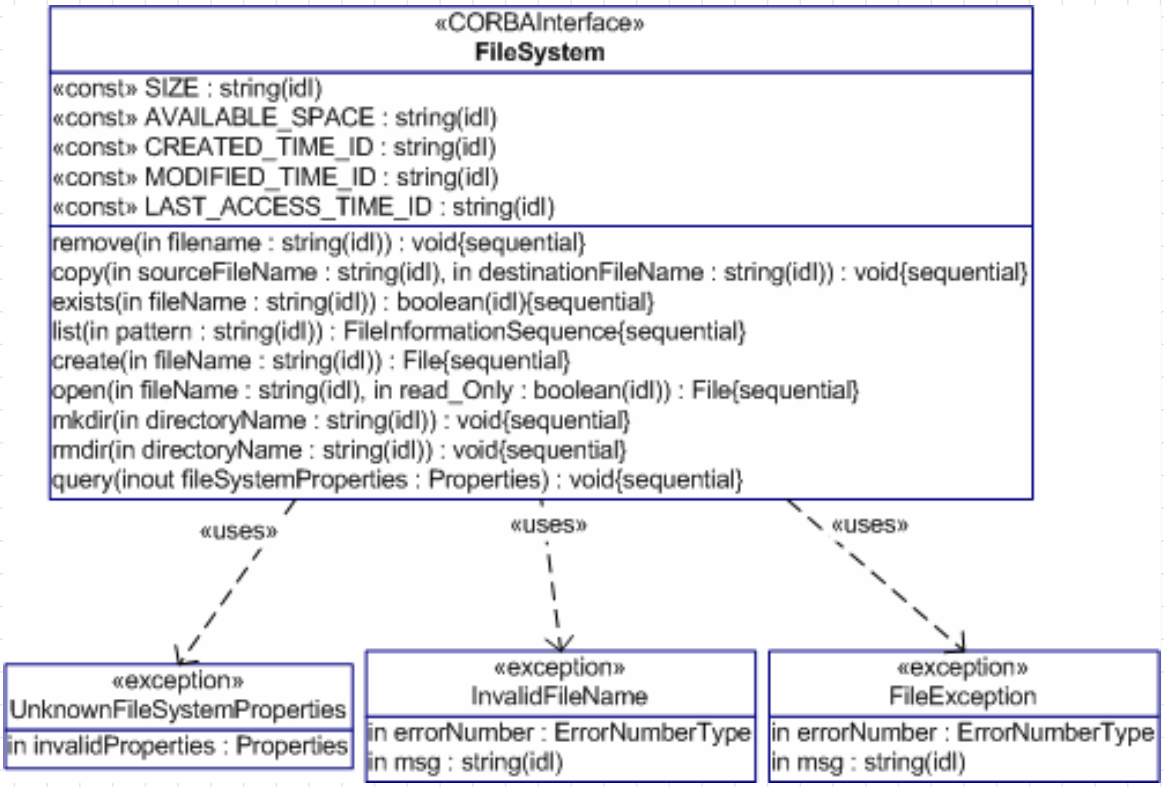
The *FileSystem* interface abstracts away where the file system object resides within the system (red-side, black-side, local, or remote).

It provides the facility of passing around a logical Network File Systems (NFS) objects as CORBA object references within the system.

The *FileSystem* interface also was chosen over a regular NFS since this may not be resident on all nodes (platforms) within a system or available for a wide range of Operating Systems.

It provides basic file system operations one would expect on file system. The behavior of these operations resembles POSIX operations.

Framework Services Interfaces: *FileSystem*



FileSystem Constants

Constants are defined to be used for the query operation and to retrieve values from `fileProperies`.

```
const string SIZE = "SIZE";
const string AVAILABLE_SPACE = "AVAILABLE_SPACE";
const string CREATED_TIME_ID = "CREATED_TIME";
const string MODIFIED_TIME_ID= "MODIFIED_TIME";
const string LAST_ACCESS_TIME_ID = "LAST_ACCESS_TIME";
```

For time properties, the identifier is a constant string and the value shall be an unsigned long long data type containing the number of seconds since 00:00:00 UTC, Jan. 1, 1970. E.g:

A value of 1411198023 corresponds to:

GMT: Sat, 20 Sep 2014 07:27:03 GMT
Local time zone: 9/20/2014, 9:27:03 AM GMT+2

FileSystem Operations

```
void remove (in string fileName) raises (FileException,
InvalidFileName);

void copy (in string sourceFileName, in string
destinationFileName) raises (InvalidFileName,
FileException);

boolean exists (in string fileName) raises
(InvalidFileName);

File create (in string fileName) raises (InvalidFileName,
FileException);
```

The `create` operation shall create a new `File` based upon the input fileName parameter.

```
File open (in string fileName, in boolean read_Only) raises
(InvalidFileName, FileException);
```


FileSystem Operations: `list`

```
FileInformationSequence list (in string pattern) raises
(FileException, InvalidFileName);
```

The `list` operation provides a list of files along with their information in the file system according to a given search pattern, which identifies one file or for a set of files.

Patterns include "*" and "?" wildcard characters used to match any sequence of characters and any single character, respectively. These wildcards shall only be applied following the right-most "/" character in the pathname contained in the input pattern parameter.

The `list` operation shall return a `FileInformationSequence` for files that match the search pattern.

The `list` operation shall return a zero length sequence when no file is found which matches the search pattern.

FileSystem Operations: `list`

`list` returns a `FileInformationSequence` defined as below.

The `FileInformationType` indicates the information returned for a file. At a minimum, the file system shall support name, kind, and size information for a file. Examples of other file properties that may be specified are created time, modified time, and last access time.
The `FileType` indicates the type of file entry. A file system may have PLAIN or DIRECTORY files and mounted file systems contained in a FileSystem.

```
enum FileType {PLAIN, DIRECTORY, FILE_SYSTEM};

struct FileInformationType {
  string name;
  FileType kind;
  unsigned long long size;
  Properties fileProperties;
};

typedef sequence<FileInformationType> FileInformationSequence;
```


FileSystem Operations

```
void mkdir (in string directoryName) raises
(InvalidFileName, FileException);

void rmdir (in string directoryName) raises
(InvalidFileName, FileException);

void query (inout Properties fileSystemProperties) raises
(UnknownFileSystemProperties);
```

The query operation retrieves information about a file system given `fileSystemProperties`' ID and at least for:

- SIZE - an ID value of "SIZE" causes the query operation to return an unsigned long long containing the file system size (in octets).
- AVAILABLE_SPACE - an ID value of "AVAILABLE_SPACE" causes the query operation to return an unsigned long long containing the available space on the file system (in octets).

Framework Services Interfaces: *FileManager*

Multiple, distributed file systems may be accessed through a File Manager.

The *FileManager* interface appears to be a single file system although the actual file storage may span multiple physical file systems.

This is called a federated file system. A federated file system is created using the `mount` and `unmount` operations. Typically, the Domain Manager or system initialization software will invoke these operations.

The *FileManager* also extends the *FileSystem* interface by providing `mount` and `unmount` behavior like a Network File System (NFS).

If a client does not need to mount and unmount file systems, it may treat the File Manager as a file system by CORBA widening a `FileManager` reference to a `FileSystem` reference.

Framework Services Interfaces: *FileManager*

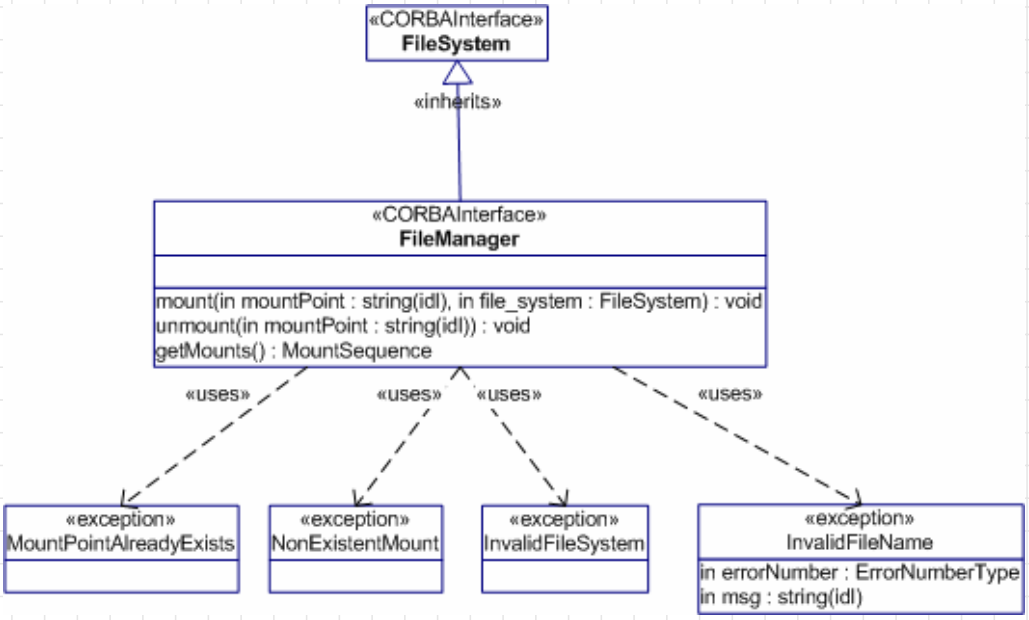
The *FileManager* interface provides a distributed file manager service capability that is used when accessing a system's (*DomainManager*) file systems.

The interface can also be used for a *DeviceManager* implementation for its file system attribute when a node has multiple file systems.

A single File Manager can be built such that it contains all the file systems in the system and be treated like one file system for a system.

This allows one to manage all the system files from one CORBA object.

Framework Services Interfaces: *FileManager*



Framework Services Interfaces: *FileManager*

Usage examples.

Based upon the pathname of a directory or file and the set of mounted file systems, the file manager delegates the *FileSystem* operations to the appropriate file system.

Example 1: if a file system is mounted at "/ppc2", an open operation for a file called `/ppc2/profile.xml` would be delegated to the mounted file system. The mounted file system will be given the `fileName` relative to it. In this example the FileSystem's open operation would receive "`/profile.xml`" as the `fileName` argument.

Example 2: when a client invokes the copy operation, the file manager delegates the operation to the appropriate file systems (based upon supplied pathnames) thereby allowing copy of files between file systems.

FileManager Operations

```
void mount (in string mountPoint, in FileSystem file_System)
raises (InvalidFileName, InvalidFileSystem,
MountPointAlreadyExists);

void unmount (in string mountPoint) raises
(NonExistentMount);

MountSequence getMounts();
```

The getMounts operation shall return a MountSequence that contains the file systems mounted within the file manager.

```
struct MountType
  {
    string mountPoint;
    FileSystem fs;
  };

typedef sequence <MountType> MountSequence;
```


Domain Profiles & Ossie

A presentation of the Domain Profiles will be provided along with a set of slides on the Ossie environment.