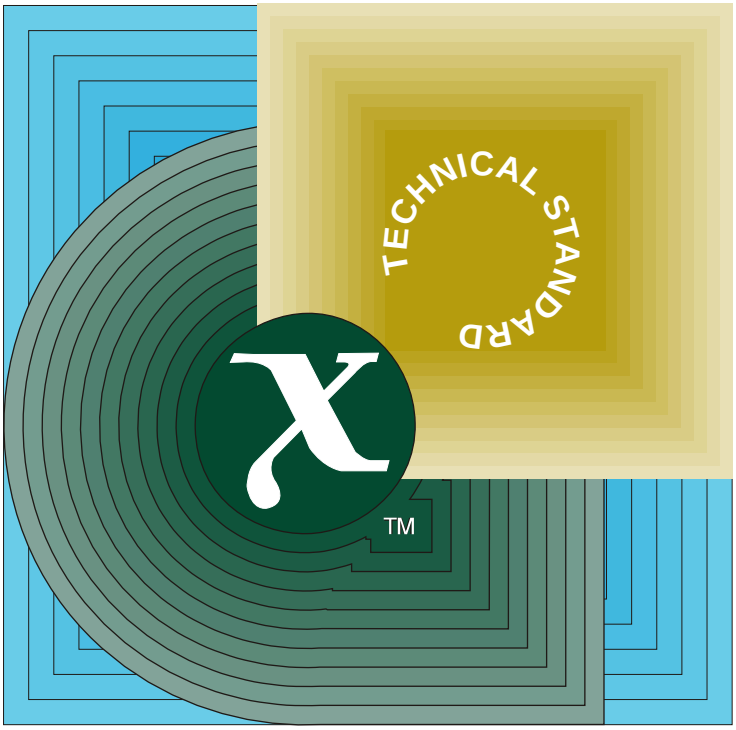
Technical Standard

Systems Management:
Common Management Facilities (XCMF)



THE Open GROUP

[This page intentionally left blank]

CAE Specification

Systems Management:
Common Management Facilities

The Open Group

This specification is published by The Open Group under the terms of its joint publication agreement with the Object Management Group (OMG).

The OMG Document Reference is *formal/97-09-32*.

CAE Specification

Systems Management: Common Management Facilities

ISBN: 1-85912-174-8
Document Number: C423

Published in the U.K. by The Open Group, October 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the ''X'' mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of the IT DialTone to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The ''X'' mark is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This specification describes an approach to the development of standards-based, open system administration applications, using common management facilities (or services).

It defines a framework (object request broker and management facilities) to allow the development of applications that will significantly decrease the effort required to administer distributed systems. The framework is based upon an industry-standard object request broker (implementations of CORBA 1.2) and Object Service Specifications.

In addition, a set of management facilities are defined that allow management-specific interfaces to be common across environments, allowing the development of heterogeneous, interoperable applications. These management facilities are key to building a foundation such that systems management applications and objects may be defined and built that span multiple vendors' framework implementations.

Audience

This specification is directed towards systems management architects, analysts, and programmers who wish to provide systems management services in an open framework based on the distributed systems management reference model (see reference **XRM**) and its components. This framework is important because it promotes application interoperability across platform types and provides a common object model for application development. It employs object-oriented programming.

It is assumed that the reader of this specification is familiar with the work of the Object Management Group (OMG). Specifically, this should include the Common Object Request Broker: Architecture and Specification (see reference **CORBA**), and the Object Management Architecture (see reference **OMAG**).

Structure

This specification is organized as follows:

- **Chapter 1** explains the scope and purpose of this specification, and identifies its relationship to the OMH Object Model and to other systems ,anagement issues.
- **Chapter 2** describes the architectural framework of the specification.
- **Chapter 3** provides the specification of the Management Facilities.
- **Chapter 4** describes the command line interfaces that allow the services to be invoked from within shell scripts.
- **Appendix A** provides IDL interface definitions that can be compiled for the systems management interfaces defined in this specification.
- **Appendix B** provides an inheritance diagram for the Common Management Facilities described in this specification.

A **Glossary** and **Index** are also provided.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members, language-independent names, and symbols that are specific to the management services interfaces in this specification.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — Function parameters, or variable names
  - — Environment variables
  - — Utility names
  - — External variables.
- The notation [ABCD] is used to identify a return value ABCD, including if this is an error value.
- Syntax and IDL code examples are shown in **Helvetica Bold** font.

Trademarks

OMG® and Object Management® are registered trademarks of the Object Management Group, Inc.

Motif®, OSF/1®, and UNIX® are registered trademarks and the IT DialTone™, The Open Group™, and the ‘‘X Device’’™ are trademarks of The Open Group.

Acknowledgements

The Open Group acknowledges the work of the Object Management Group's (OMG) Revision Task Force (RTF) on the Common Management Facilities (XCMF) Specification. The OMG adopted the X/Open XCMF Preliminary Specification (P421) as their base document, developing it to produce their OMG XCMF Specification. The active members of the OMG RTF were:

Zoely Canela	Alcatel
Michael Greenberg (Chairman)	NEC
Jim Hughes	Fujitsu
Russell Newcombe	IBM
Jim Willits	HP

This XCMF CAE Specification from The Open Group is fully aligned with the OMG's XCMF Specification.

Referenced Documents

The following documents are referenced in this specification:

CORBA 1.2
: CAE Specification, July 1994, The Common Object Request Broker: Architecture and Specification (ISBN: 1-85912-044-X, C432), in conjunction with the Object Management Group (OMG).

DTP
: Guide, February 1996,, Distributed Transaction Processing: Reference Model, Version 3 (ISBN: 1-85912-170-5, G504).

Internationalisation Guide
: Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304).

OMAG
: OMG (Object Management Group) Object Management Architecture Guide (OMA Guide), Version 2.0, September 1992. Published by OMG, 492 Old Connecticut Path, Framingham, MA 01701, USA.

OMGOM
: OMG (Object Management Group) Object Model (OM). Included as Chapter 4 of reference **OMAG**, but being separately revised at the time of publication of this Common Management Facilities Volume 1 Preliminary Specification.

COS, Volume 1
: Preliminary Specification, July 1994, Common Object Services, Volume 1 (ISBN: 1-85912-482-2, P432), in conjunction with the Object Management Group (OMG).

RDSMF
: UNIX International System Management Working Group, Requirements for the Distributed System Management Framework, April 20, 1991.

XA
: CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification (ISBN: 1-872630-24-3, C193).

XPG3
: X/Open Specification, 1988, 1989, February 1992 (ISBN: 1-872630-43-X, T921); this specification was formerly X/Open Portability Guide, seven volumes, January 1989 (ISBN: 0-13-685819-8, XO/XPG/89/000).

XPG3 Overview
: X/Open Portability Guide, February 1992, XPG3 Portability Guide Overview, Issue 3 (ISBN: 1-872630-44-8, X200).

XPG4
: X/Open Systems and Branded Products: XPG4, October 1994 (ISBN: 1-872630-52-9, X924).

XPG4, Version 2
: The X/Open Branding Programme, How to Brand — What to Buy, February 1995 (ISBN: 1-85912-084-9, X951).

XPS
: Snapshot, 1991, Systems Management: Problem Statement (XO/SNAP/91/010 or S110).

XRM
Guide, August 1993, Systems Management: Reference Model (ISBN: 1-85912-05-9, G207).

Chapter 1

Introduction

1.1 Overview

The Systems Management Reference Model (see reference **XRM**) consists of 3 basic components:

- *Managers* which implement Management Tasks and other composite management functions.
- *Managed Objects* which encapsulate resources. Resources are the entities within a system or network of systems that require management.
- *Services* which provide the System Management (XSM) Support Environment. The XSM Support Environment consists of the capabilities and interfaces that are necessary to support the other components of the Reference Model.

Management Facilities are a category of services which have been specialized for XSM distributed systems management. This specification defines a set of management facilities that supplement the Object Management Group's (OMG) Object Model so that it supports the System Management Reference Model. The Systems Management Reference Model (see reference **XRM**) provides a complete description of the mapping to the OMG Object Model.

1.2 Relationship to the OMG Object Model

The OMG has developed a conceptual model, known as the core object model, and a reference architecture upon which applications can be constructed (see references **OMAG** and **OMGOM**). The OMG OMA defines the composition of objects and their interfaces.

A fundamental architecture of the OMA is the Common Object Request Broker Architecture (see reference **CORBA**) that specifies a framework for transparent communication between application objects. The Object Request Broker (ORB), a key component of this architecture, provides the mechanisms for issuing requests to objects and returning responses.

In addition to CORBA, the OMG has also published a set of Object Services that are common to a wide range of application domains. The first set of these services (see reference **COS Volume 1**) includes:

- Life Cycle Services
- Naming Services
- Event Services

Life Cycle services define interfaces for creating, deleting, moving, and copying objects. Naming Services specify interfaces for binding and resolving names. The Event Services provides mechanisms for decoupled communication between objects, supporting both a push and a pull model of communication.

In addition to these services, additional activity is underway in the OMG to adopt specifications for Concurrency Services, Externalization Services, Persistent Storage Services, Relationship Services, Transaction Services, Time Services, and Security Services. Work is also coming to completion on a Licensing Service, a Properties Service, and a Query Service.

It is the implementation of the OMG architectures and Object Services that constitute an *OMG environment*. This environment provides much of the necessary infrastructure for supporting distributed system management.

1.3 Scope of this Specification

This specification presents a set of management services that integrate with the OMG environment and provide extended services specifically for the distributed systems management. These services, in conjunction with the OMG environment, are fundamental to provide a framework for developing distributed systems management applications.

The management facilities specified assumes an *OMG CORBA 1.2*-compliant ORB and a compliant implementation of the Common Object Services (see reference **COS Volume 1**). This implies the management facilities described in this specification may use types and interfaces defined in OMG standard header files (for example, **<orb.idl>**).

The components addressed in this specification are those focused on the management of policy-driven objects including the mechanisms and facilities that enable the establishment and enforcement of policy on these objects. The Reference Model is used in Figure 1-1. to illustrate the focus of this specification. The Object Request Broker and Object Services discussed in this specification are drawn from the OMG environment.

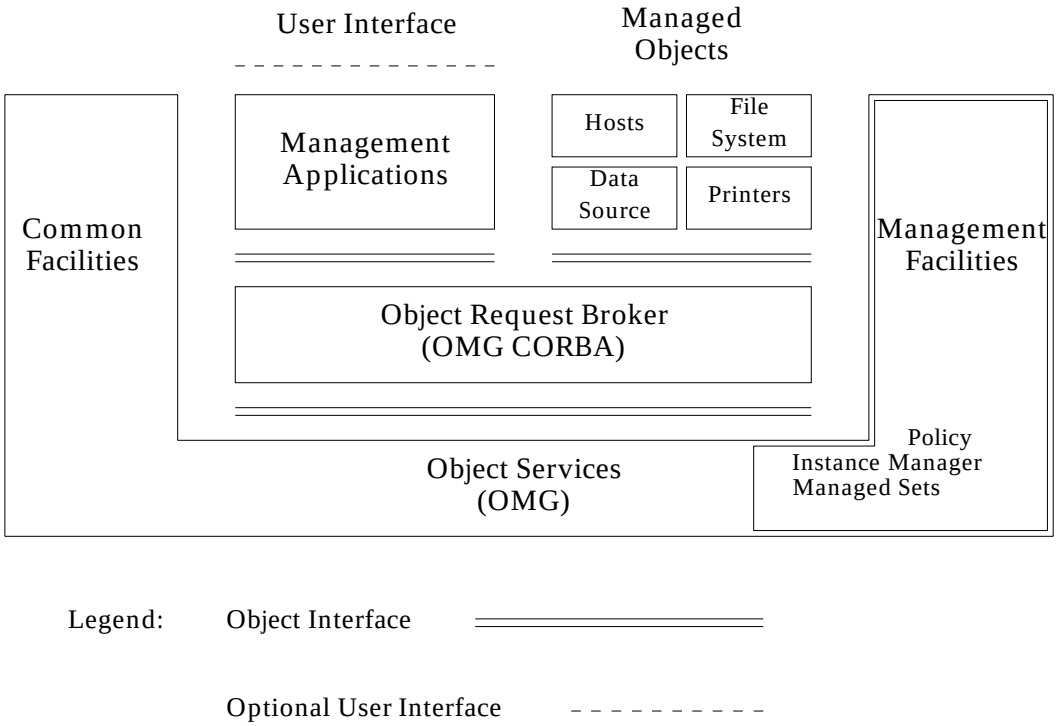


Figure 1-1 Systems Management Framework Components

The systems management application domain is a vertical application market with specialized requirements. The work OMG is performing at the ORB and Object Services level is common across many, if not most, application spaces. Systems management requirements exist today and a considerable industry is beginning to build applications in this area. Thus, there is a need to standardize interfaces for systems management. This specification does not supersede the OMG Object Services. Rather, it complements the OMG Object Services by defining interfaces that are fundamental for developing distributed system administration applications.

This specification also fully backs application portability and internationalization objectives. In areas where relevant standards have been identified (see referenced documents), these standards are used. Examples are the Portability Guide, Issues 3 and 4 (see references to **XPG** documents, and to reference **Internationalization Guide**). Adhering to these specifications is critical to all implementations and the interfaces for a system administration framework must enable the use and accommodation of these specifications.

1.4 Components Not Addressed

This section outlines a set of application development issues that are not addressed in this specification, but that must be addressed during the development of system administration applications.

1.4.1 Security

Providing a robust and flexible security service is crucial for the development of distributed management applications. Providing such a service in a heterogeneous and distributed environment is a very complex undertaking. The OMG is currently developing a Security service for the CORBA environment. The Open Group is tracking its progress closely and hopes to build upon this service. Because of the complexity of defining the service and the ongoing working in the OMG, a Security service is beyond the scope of this Specification.

1.4.2 Graphical User Interface/Desktop

There are other standards groups and industry consortia working to define a common desktop environment, including the graphical user interface (GUI) technology. Proposing a particular approach to this effort through a parallel process is inappropriate. While the GUI and desktop is critical to the success of individual administration applications, it is not fundamental for the development of distributed management applications.

1.4.3 Application Specific Resource Interfaces

This specification details a set of interfaces for management facilities that enables the development of distributed management applications. These applications will include policy-driven objects that encapsulate managed resources. The definition of the interfaces to the objects to encapsulate managed resources is outside the scope of this specification.

1.5 Interoperability Issues

Interoperability is key to building distributed systems management applications. Distributed systems management applications using the OMG environment requires interoperability at two distinct levels:

- **ORB-to-ORB**

  Interoperability at this level requires that ORBs developed by two independent entities are able to communicate with each other. The method requests dispatched by one ORB are able to be received and understood by another.

  ORB-to-ORB interoperability is addressed by OMG's CORBA 2.0 Specification.

- **Object Interfaces**

  Interoperability of object interfaces requires that a objects developed in one OMG environment have an interface (or subset of inherited interfaces) that is common to another environment. That is, an object developed by one vendor may make a request on an object developed by a different vendor through a known interface. This is true for both compiled, executable applications development as well as for implementations of policy by the system administrators.

  Object Interfaces interoperability is being addressed by the OMG in the Object Services. These object service interface specifications address a very general class of distributed, object-oriented applications. The management services interfaces detailed in this specification define interfaces for the development of system administration applications. Thus, this specification provides interfaces and command line interface equivalents that are specific to developing interoperable system administration applications integrated with the OMG environment.

  Interoperable object interfaces also allow objects to be portable across different implementations of the OMG environment.

1.6 Future Directions

The facilities specified in this specification provide only a subset of all the facilities necessary to build portable and interoperable management applications. As CORBA based management frameworks become more prevalent in the industry, more management facilities will be developed and deployed. The developers of this Specification recognize this fact and hence expect to identify and standardize additional management facilities in future.

Some number of facilities and language bindings were identified as important during the development of this Specification, but because of resource limitations are not included in this Specification. Some of the facilities identified were such things as process management, scheduling, event management, and a Perl language binding. These facilities are excellent candidates for inclusion in a future volume of management facilities. However, this vision should in no way be interpreted as limiting the breadth of future submissions in this area.

Chapter 2

Management Facilities Architecture

This chapter describes the management facilities for developing system administration applications. These management facilities use and build on the CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve, and specialize functionality

The following sections provide a detailed description of how the interfaces and functionality combine to form the basis for system administration applications.

2.1 Managed Set Service

The object model supports the concept of sets of managed objects. *Managed* sets organize objects into groups. The **Set** and **Member** interfaces support the basic set functionality. This functionality can also be satisfied by other, more general, relationship services. The **Set** interface defines the operations required for one object to maintain a reference to each object it contains. The **Member** interface defines operations that allow an object to maintain references to objects by which it is contained. Managed sets provide a many-to-many relationship.

In the set relationship, all member objects know of all sets to which they are members, and all sets know of all objects which are members of the set. Each participant in the relationship maintains a list of the other participant's object references. Objects supporting the **Member** interface contain *back references*. A *back reference* is a reference from an object to the set object to which the object belongs.

Sets may be subject to policies, in that an object can inherit both the **Set** and **PolicyDrivenBase** interfaces. The decision about whether a set is policy-driven is independent from the decision of whether to associate policy with the members of a set. A set may be subject to policy, and its members may not.

Set objects contain references to a set of objects, which are referred to as it's members. There is no unified ordering to the members of a set. Objects must support the **Member** interface in order to be allowed as members of a set. The **Set** interface itself inherits the **Member** interface. Therefore, set objects can be members of sets.

An example of a set could be the users within a system who are members of the company volleyball team, or all users that receive mail regarding company marketing plans. Sets offer a way for system administrators to organize resources into logical groups.

Because sets can refer to other sets, they can be organized into hierarchies. In addition, because sets are just groups of references to objects, an object can belong to any number of sets. Users of an application can organize their managed resources into a suitable hierarchy, and create multiple sets of those resources. A set can be heterogeneous, meaning that it can group objects of different types. Application developers can create sets that have as members a combination of hosts, users, or other types of objects.

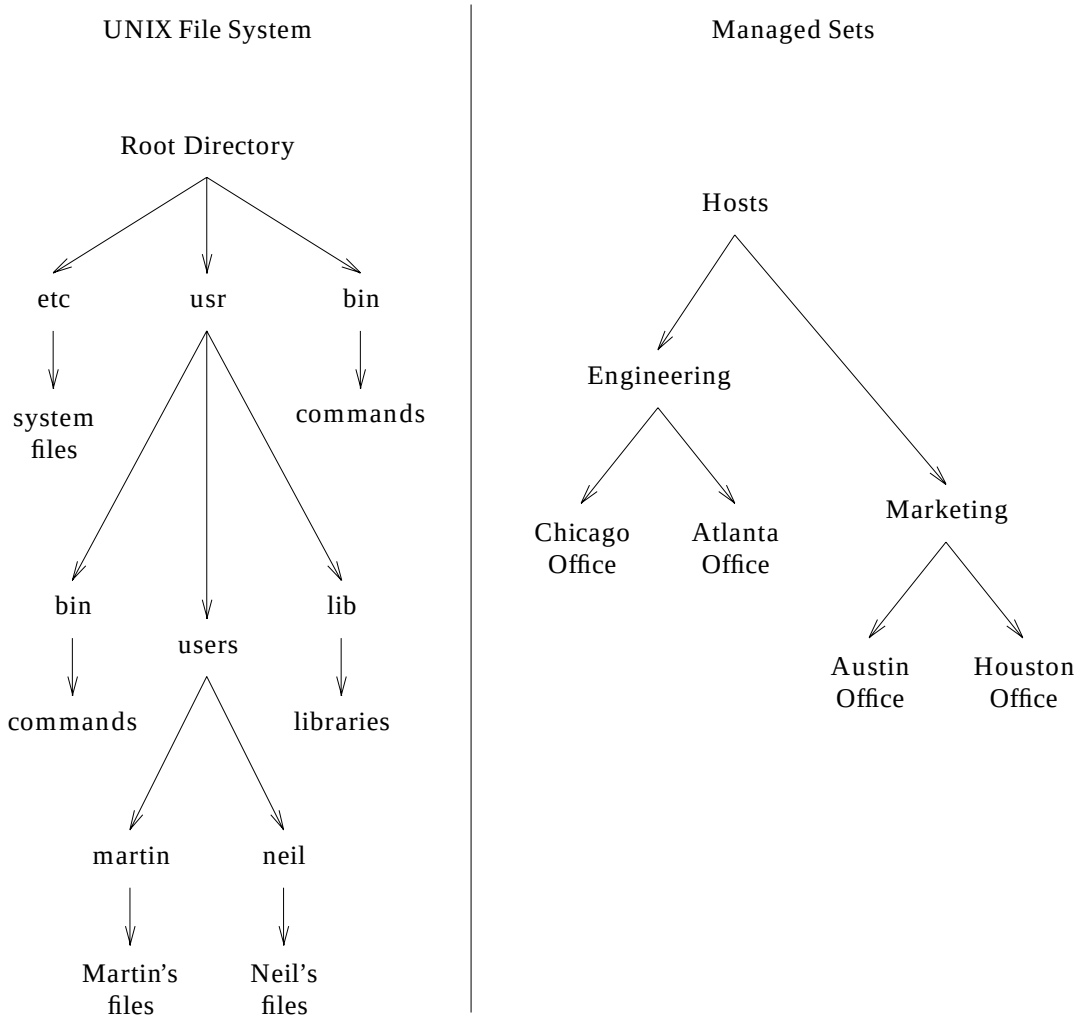


Figure 2-1 Managed Sets Compared with UNIX File Systems

Figure 2-1 illustrates the parallel nature of managed sets and UNIX directories. Both sets and directories contain various types of items, including subsets and subdirectories. You can think of a set as a directory and an object as a file. Note that the figure shows the Hosts set object as an example of an set that is not a member of any other set. In this way it is similar to the root directory of a file system. As shown in the figure, the Hosts object contains the Engineering and Marketing objects, which contain the hosts maintained by these departments. The parallel to a UNIX file system is not exact, however, because you can remove a set without deleting its members. In fact, you can remove the last set to which an object belongs without deleting the object.

Because they inherit the **Member** and **Set** interfaces, operations invoked on an object supporting the **Set** interface will not propagate along the relationships of the set. That is, when a set is removed, the objects that are members of the set are not affected.

2.2 Policy-driven Base Service

The **PolicyDrivenBase** interface provides common operations on all policy-driven objects within a system administration application. The operations are a common set of operations that have been grouped within a single interface. These common operations are a set of behaviors that allow objects to be managed by policy regions. This interface is the basis for application development in the systems management framework when dealing with policy-driven objects.

The operations defined in the **PolicyDrivenBase** interface are the minimum required to support object-oriented policy-driven application integration and installation in the framework. Without such a capability each application and policy-driven object within an application would not be able to be integrated except in a superficial manner. This service provides a strong foundation that enables applications to be more fully integrated and function in a similar manner.

The **PolicyDrivenBase** interface inherits the **ManagedInstances::Instance** interface. Thus, all policy-driven base objects exhibit the behaviors of managed instances, such as having a name, being capable of belonging to sets, and being managed by an instance manager.

The **PolicyDrivenBase** interface is to be inherited by policy regions and by any object that is to be a member of a policy region. A policy region is a entity that allows a system administrator to associate custom rules or policies with objects in an installation.

2.3 Instance Management Service

The systems management framework is implemented in an OMG environment. In an OMG environment, there is no explicit support for creating and managing objects for the systems management domain. However, for developers of management applications it is useful to use traditional object-oriented concepts to manage and create managed objects. The *instance management* service provides basic object creation and management capabilities for all types of managed objects, including policy-driven and representative types.

The *Instance Management Service* defines interfaces for creating and managing object instances within the systems management framework. This service is provided by the **ManagedInstances** module and defines three fundamental roles; an object can be:

- A managed instance
- A factory for and managed set of a specific type of managed instances (instance manager)
- A factory for and managed set of instance manager objects (library)

A *managed instance* is an instance of a particular type of managed object that is represented and managed by a single instance manager. A managed instance object supports the **Instance** interface. The **Instance** interface provides operations that determine and report the object's type and return the object reference of the instance manager by which it is managed.

An instance manager acts as a factory for managed instances, encapsulating the type and implementation specific information needed for managed object creation. There may be many managed instances of a given type within an installation. Additionally, there may be one or more instance managers that manage the instances of a given managed object type. The **BasicInstanceManager** interface defines operations needed to create managed instances and group them into sets. The **InstanceManager** interface provides all capabilities of the **BasicInstanceManager** and the additional capability to support the specification of policy to be associated with managed instances.

A library acts as a factory for creating instance managers and has the ability to maintain a list of instance manager objects. A library also acts as a factory finder, allowing instance managers to be selected based on characteristics such as the interface of managed instance that they support and the type of policy objects that are registered with them. The creation and deletion of instance manager objects through the library has the effect of adding and removing managed object types to the running environment. In a given installation, there may be more than one library object maintaining instance manager information.

2.3.1 Instances

The **Instance** interface provides the fundamental operations that are needed for an object to be managed by an instance manager. Through inheritance and newly introduced operations, the **Instance** interface provides a set of generic, low-level behaviors that allow objects to be managed, grouped and named.

The **Instance** interface inherits the **ManagedSets::Member** interface. This is fundamental to allowing managed instances to be managed by instance managers. It also provides the **Identification::Labeled** interface which allows a managed instance to be named, which instance managers require of their instances.

The **Instance** interface introduces new operations which can report information about the managed object's type, who its instance manager is and the location where the managed object exists.

2.3.2 Basic Instance Managers

A *basic instance manager* object exists for each defined managed object type. The basic instance manager encapsulates implementation specific details of a managed object type. Creation of managed objects is one of the important implementation specific details. A basic instance manager can report some of this information by returning the **InterfaceDef** for the type it manages. In order for a type to be managed by the **BasicInstanceManager** interface, it must support the **Instance** interface.

It is important to note that the relationship between managed instances and basic instance managers is one-to-many — one instance manager for many instances. The relationship between managed object types and basic instance managers is also one-to-many — potentially many basic instance managers supporting a single managed object type.

This is illustrated in Figure 2-2. Because there may be more that one basic instance manager per managed object type, basic instance managers could also represent a scope of influence, and manage the object types within their scope.

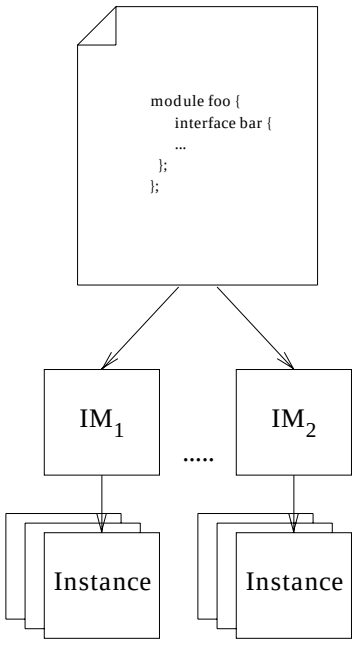


Figure 2-2 Managed Object Types Relationship to Basic Instance Managers

Each managed object type controlled by a basic instance manager can be identified by both a type name and a type definition. The type definition is specified in the IDL interface defined by a programmer and is contained in an **InterfaceDef**. The type name should be a name suitable for display at the user interface and for use as an argument in command line operations and programs. The basic instance manager maintains the type definition and the type name. The type name is available on a read-only basis from each managed instance of the type.

As a participant in object creation, a basic instance manager encapsulates much of the behavior commonly considered *factory object* and *factory finder object* behavior. A factory can create objects. A factory finder locates a factory for a particular type of object in a particular location. In the general case, a factory finder can be used to locate a factory that can create objects of *type X* in *location Y*. The factory behavior of the basic instance manager is exposed to the client through

the **CosLifeCycle::GenericFactory** interface, while the factory finder behavior is not exposed to the client for direct use.

The client's view of object creation is depicted in Figure 2-3. When a client creates a managed object, the client locates the basic instance manager associated with the managed object type to be created. The basic instance manager can be located using the library object. The client invokes the *create_object* operation, inherited from the OMG standard **CosLifeCycle::GenericFactory** interface, on the instance manager object. This operation provides a common interface for creation of all managed objects. The *create_object* operation is a specialization of object creation as defined in the OMG Life Cycle Service. When invoking the *create_object* operation, the client specifies a label for the new managed instance and the desired location of the managed instance identified by a location object reference. These parameters are passed through the *the_criteria* parameter of the *create_object* operation. If each of these parameters is not specified, an exception is raised. The label and location are a specialization of object creation as defined in the OMG Life Cycle Service.

The Common Object Services Specification (see reference **COS Volume 1**) defines the *k* parameter of the *create_object* operation (of type **Key**) as being used to identify the desired type of object to be created. This definition, while interesting in a more general application domain, is redundant with the concept of instance managers. Thus, the specification of the **Key** in the *k* parameter is not required, and may be ignored by implementations of the *create_object* operation of the **GenericFactory** interface when inherited by the **BasicInstanceManager** interface. Alternatively, implementations may use this information, if supplied by clients of creation, as a basic form of run-time type checking to ensure that the basic instance manager really creates the managed object type the client desires.

When a *managed instance* is created, the requested label must be verified for uniqueness. The label that is stored within the state of each managed instance is actually comprised of two fields: *id* and *kind*. The *id* field is set to the string passed by the client of object creation, while the *kind* field is filled in by the **BasicInstanceManager** to uniquely identify the **BasicInstanceManager/Library** object pair that was responsible for creating the instance. **BasicInstanceManagers** are responsible for enforcing the fact that the value for the *id* field supplied by the client is not already in use by a managed instance contained within the same **BasicInstanceManager**. In this way, the two-component label of any object is unique among all managed instances created within the environment. If the requested label id is already used by a managed instance contained by the **BasicInstanceManager** upon which a creation request was performed, an exception is returned and the creation fails. Note that since the label type is really the **NameComponent** type defined in the **CosNaming** module, usage of the COS Naming Service (see reference **COS Volume 1**) to maintain consistency between the managed object namespace and managed object containment hierarchy is straightforward[1].

The *location* of the *managed instance* is specified by supplying an object reference to an object supporting the **Location** interface. This interface defines the logical location for the managed instance. The location could be a set of hosts in an environment, or it could be a specific machine. In system administration applications it is critical to be able to locate a managed instance on a particular host. For example, if a managed instance represents a user account, then when the account information changes, the files on exactly one specific machine should be changed to reflect the new information.

---

1. In fact, a possible implementation of the **ManagedSet** service would be to wrapper an implementation of the **CosNaming** service.

Any type-specific initialization for a newly created object is handled by interactions between the basic instance manager and the factory object. This design allows a client to treat managed objects of all types identically for object creation, yet allows object-type specific interfaces to be used with the actual operation on object factories. Therefore, to the client a managed object creation request simply involves invoking a standard request on a basic instance manager and having the object reference of the new object returned.

Type- or implementation-specific initialization may involve creating or allocating persistent storage, defining default values for attributes or states, or involving the new managed instance in relationships with other objects. When a managed instance is created, regardless of the type, it becomes a member of the basic instance manager set. This set serves as a repository that may be queried to find managed instances of a given type. Other name spaces for managed instances may be created for performance or organizational reasons. However, this default set ensures that managed instances are not orphaned, and serves as a starting point for building alternate name spaces.

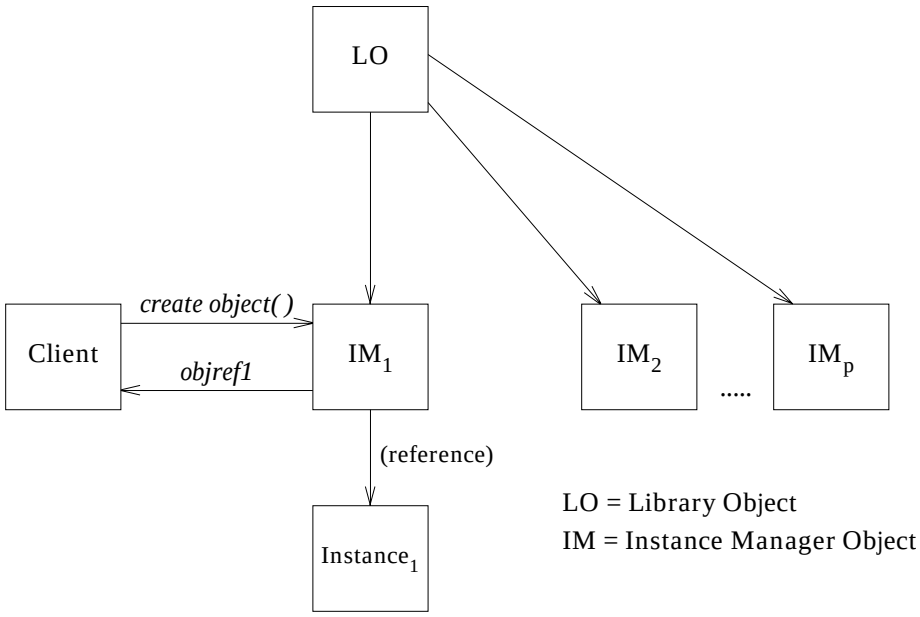


Figure 2-3 Client View of Object Creation

The basic instance manager performs all required work to satisfy the request for managed object creation. The basic instance manager finds a factory of the correct type, in the correct location. Once the factory is located, the basic instance manager can enter into a private protocol with the factory to provide object type- or implementation-specific initialization or other information. The basic instance manager can obtain the object reference of a factory in a particular place in any of a variety of ways. For example, by querying a name service, if one is present, or it can maintain a private copy of the object references to factories on other client machines or servers using the location object reference.

Because of their use for object creation and the need to interact with the actual object factories for creation of managed objects, much of the implementation of a basic instance manager will normally be tied to a specific ORB implementation. Since a basic instance manager deals with only one type of object, the **InterfaceDef** used when registering managed instances with the basic object adapter (BOA) will be the same for all instances. The *ImplementationDef* used may vary across instances based on the host or server in which the managed instance is created. The

information needed by a basic instance manager at the time it is created and installed in an environment will be dependent upon the needs of the ORB implementation to which it is related.

2.3.3 Instance Managers

Objects that support the **InstanceManager** interface provide all of the capabilities that the basic instance managers provide, and in addition provide operations and behavior that allow it's managed instances to be subject to policy management. Instances managed by an instance manager must support the **PolicyRegions::PolicyDrivenBase** interface.

The *create_object* operation on an instance manager is a further specialization of what is defined in the OMG Life Cycle Service. In addition to the specializations defined for the basic instance manager, the instance manager requires that one or more policy regions be passed in the *the_criteria* parameter of the *create_object* call. The policy-driven base managed object that is created will be added as a member of each of these policy regions.

Instance managers provide operations for the registration of *initialization policy objects* and *validation policy objects*. These policy objects are specifically associated with the type of policy-driven base managed object supported by the instance manager. These are the policy objects that are available to policy regions for use with this type of managed object.

The initialization policy objects and validation policy objects are used by policy regions to encapsulate a set of management policies and support high-level operations that control a particular managed object type. Initialization policy objects support operations to define the initial (default) values of the policy-driven object attributes. Validation policy objects support methods that can validate initial values or changes to object attributes, and also methods that can be used to control object behaviors. In Figure 2-4, the Library object is shown referencing an instance manager and several basic instance manager objects.

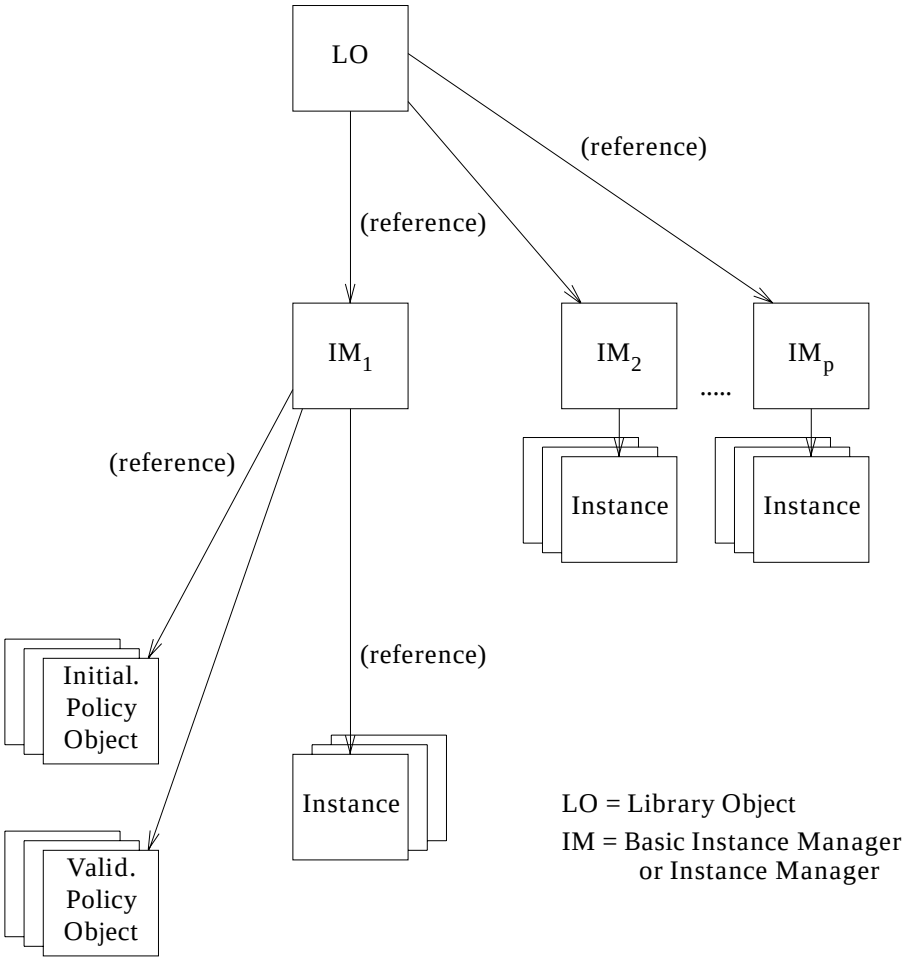


Figure 2-4 Instance Managers and Policy Object Relationships

A single instance manager object may contain references to one or more initialization policy objects and one or more validation policy objects. Each such policy object can define different policies that can be used to control managed instances. Users can define their own management policies by customizing an implementation for an existing initialization policy or validation policy object, or by creating a new implementation of an initialization policy or validation policy object and attaching the new object to the instance manager.

In general, throughout this specification, the use of the terms *instance manager* and *basic instance manager* is not precise. A reference to a *basic instance manager* describes all instance managers, and reference to an *instance manager* may well also apply to basic instance managers, unless the subject matter specifically deals with policy management.

2.3.4 Library

The **Library** interface introduces no new operations, but performs three major functions through the interfaces that it inherits. First, through the inheritance of the **CosLifeCycle::GenericFactory** interface, a library acts as a factory object for the creation of the various kinds of instance manager objects. Through the inheritance of the **ManagedSets::FilteredSet** interface a library is able to track and manage the instance manager objects that it creates. Finally, through the inheritance of the **CosLifeCycle::FactoryFinder** interface, the library performs the service of locating instance managers for clients.

When a new instance manager is developed, it is installed into an environment when it is created by a library object. The *create_object* method used to create the instance manager is a specialization of object creation as defined in the OMG Life Cycle Service. There are several different elements that can be passed on the *create_object* call using the *the_criteria* parameter (these are defined in Chapter 3). Which of these elements are required, which are optional and which are not supported is implementation specific.

The *find_factories* method of the **CosLifeCycle::FactoryFinder** interface is used to locate instance managers which have certain characteristics. These include thing such as the type of objects the instance manager supports, the interface of the instance manager itself and the kinds of policy objects registered with it. Clients can use this capability to locate an instance manager which meets their exact needs.

It is anticipated that in an implementation of the system administration framework with a name service, the name(s) and object reference(s) to the library object(s) will be bound into the name service. This binding will allow objects to know very little about an installation prior to beginning to discover instance manager objects and managed objects in the installation starting at the library object.

2.4 Policy Management Service

Policies give administrators a way to customize applications to their specific needs. A policy is a rule that an administrator places on the system. For instance, a policy can determine which users belong to a group, which users have access to a certain host, or where a user's home directory must reside.

Policy makes it easier for administrators to customize applications by allowing administrators to make the applications reflect the way their systems are managed. With policy, administrators can implement their own organization-specific rules for system administration.

Like any other set of rules, policy must be enforced to be effective. *Policy regions* enable the enforcement of policy. Policy regions associate specific policies with instances of policy-driven object types. A policy region is a special type of set of policy-driven objects. Like sets, policy regions can be arranged hierarchically according to organization- and administrator-specific criteria and can contain any set of objects an administrator wants.

The policy service defines interfaces related to policy and the management of policy. These interfaces relate to the establishment of policy regions, the objects within a policy region, the reporting of associated policy, and the definition of policies themselves.

This section describes the policy management service including policy regions and policy objects.

2.4.1 Policy and Administrators

A policy region is a set of managed resources that share common management policies. As such, a policy region object is a set object that supports the *PolicyRegion* interface and has management policies associated with it. A policy-driven object must belong to at least one policy region, and the policy regions it belongs to establish the policies enforced on the object. Each policy-driven object initially belongs to the policy regions specified as part of the *create_object* operation invocation. However, policy-driven objects may be added, deleted or moved from one region to another by an administrator. Policy regions may support many types of policy-driven objects. For example, a policy region that supports user, host, and group objects would have policies defined in a user policy object, a host policy object, and a group policy object, and thus have access to methods for managing each of these types of object.

Policy regions are filtered sets to allow them to be grouped and managed in an organization that is natural to the administrator. While the selection of members for a policy region is arbitrary in terms of the framework itself, policy regions can be used to model real-world organizations. For example, an administrator could create a policy region that represents the network resources belonging to the Engineering Department. Member objects of the policy region would follow policies governed by the Engineering Department. A policy region can contain any object supporting the **PolicyDrivenBase** interface, including other policy region objects.

One of the most important requirements of policy regions is that they be easily customizable by an administrator to implement the local policy of an organization. This can be done in one of two ways: by replacing or customizing one or more operations of an existing policy object whose operations are used by the policy region object, or by creating entirely new policy objects that implement a different policy than the original. A single instance manager that supports objects of a single type may reference many policy objects, each of which implement a different kind of policy, such as an *engineering user* policy and *marketing user* policy. Only one of a given instance manager's policy objects may be associated with any given policy region.

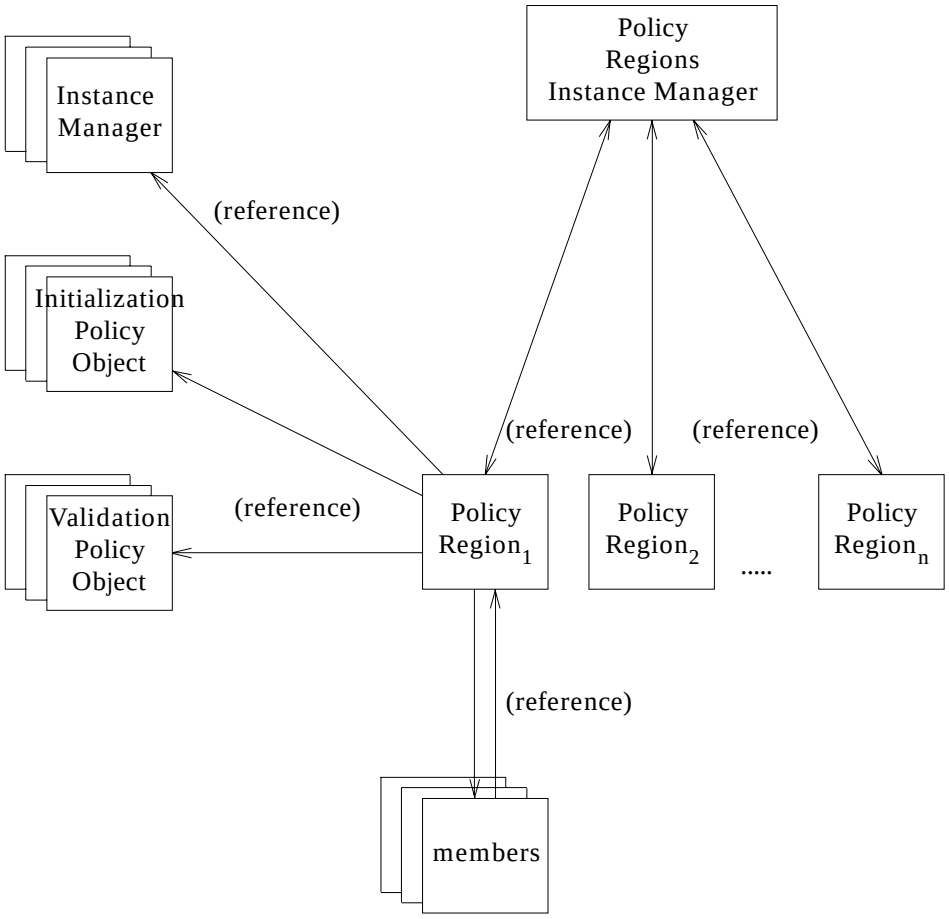


Figure 2-5 Policy Region and Policy-Driven Object Relationships

Figure 2-5 illustrates the relationships between policy regions and policy-driven objects. A policy region is shown referencing several instance managers, initialization policy and validation policy objects, and instances of policy-driven objects. Each policy region object references a set of policy-driven objects called its members. In the Engineering policy region, these objects might represent all the users and all the hosts that are part of Engineering. For each type of object supported, the policy region maintains a reference to the appropriate instance manager object and uses the methods from a single initialization policy object and a validation policy object associated with that instance manager.

In addition, this figure reflects the fact policy regions are themselves managed by an instance manager. This is labeled ‘‘Policy Regions Instance Manager’’.

2.4.2 Policy Region

Policy regions support management policies by using methods of the initialization policy and validation policy objects that are associated with object types supported by the policy region. If a type is supported, objects of that type are allowed to be members of the policy region. Supporting an object type means that:

- The policy region has a reference to the instance manager(s) for the type.
- The policy region maintains a reference to an initialization policy object for the instance manager.
- The policy region maintains a reference to a validation policy object for the instance manager.

Policy region objects supply the policy implemented by the initialization policy and validation policy objects for each supported type. The initialization policy object for a type provides a method to initialize new objects. New objects are thus always created within the context of a policy region and are said to be members of the policy region.

An initialization policy object provides methods to generate initial (default) values for the attributes of the policy-driven objects. The methods of an initialization policy object thus provide a set of recommended values to use when creating new objects. These may be modified on a per-policy region basis, thus allowing for different configurations for the same type in separate policy regions. For example, the default login shell policy for a user in the Engineering policy region might be **/bin/ksh** while the default login shell policy for a manager in the Executive policy region might be a window-shell. Note that an initialization policy object does not, however, provide enforcement as far as maintaining those values. This is performed by validation policy methods.

A policy region may also use the methods from a validation policy object for an object type. When this is true, the policy region object is said to have validation of policy enabled for that type managed by the instance manager. When validation of policy is enabled, the policy region object provides a powerful mechanism that enables enforcement of management policy. The validation policy object provides a validation method for each aspect of a managed object that is subject to policy. Validation operations enable the state of objects to be verified against policy. Object state in this context includes such things as data values stored by the object (attributes) and operational information related to the object (backup this object's state daily), etc. Thus, a broad range of components may be validated using this service. Typically, each validate operation is defined to take as input data that reflects the proposed change to the state and returns TRUE or FALSE as to whether the new value is valid. If FALSE is returned by the validate method, the update (or initial value) of the attribute is rejected and the modification request can be aborted.

2.4.3 Enforcement of Policy

The enforcement of policy may be performed in a variety of ways. The discussion presented in this specification addresses examples of mechanisms to enforce policy, and is not an exhaustive list.

The mechanism for the enforcement of policy is an application development-time procedure. The policy-driven base object designer defines which attributes, activities or methods should be subject to policy. Recall that policy is enforced by methods that encapsulate the particular policy and evaluate a proposed change. The policy method either evaluates to true or false and the requested activity is either allowed or rejected, respectively. Given that it is not possible to associate a policy with something at application run-time, it is strongly recommended that the application developer take a liberal view of what might be subject to policy. In general, all

operations that can cause a change in state of the object and the underlying system resource it models are appropriate for the application of policy. Typically, policy methods are, by default, disabled so that the policy is not checked, or the policy methods are hard-coded to return true without any further processing.

Each invocation of a policy method is preceded by a check to see whether policy has been enabled or disabled. If policy is disabled, then the policy-driven base object should be structured to act as if there was no policy associated with the activity. Again, if policy is enabled, then the policy method associated with the particular activity is invoked to determine whether to allow or disallow the operation.

Given this means for the enforcement of policy, the application developer has significant latitude in the ways in which policy is enforced by an application. For example, when an object could not be subjected to policy validation on a regular basis, it could be an administrator defined time when the policies are checked. As another example, when an object is moved from one set to another, or simply added to a set, then the set object could, prior to adding it to the set, check to verify that the object to be added conforms to the relevant policy for the set.

This discussion has focused primarily on policy validation, the verification that an object conforms to the policy to which it is subject. In addition, there are initialization policies that might be associated with objects. Initialization policies are generally associated with object creation and enable system administrators to write a method that will generate default values for parameters that might follow a simple formula, thereby saving the administrator significant time during the operation of the network and day-to-day administration. Additionally, a more senior administrator might determine the defaults to be used and thus not require or enable a more junior administrator to ‘‘make up’’ values for particular parameters.

Chapter 3

Management Facilities Specification

This chapter fully specifies the management facilities described in the previous chapters. In designing the interfaces presented in this specification, several design principles guided the activity.

- The interfaces have been developed with an eye towards the implementation on a generic ORB environment. Every attempt has been made so the interfaces, as well as their semantics, will be portable among all CORBA-compliant, ORB implementations. In the specification of the interfaces, no implementation specific features of any ORB environments are used.
- The interfaces have been designed such that the scalability of the implementation is not limited by the interface defined in IDL.

  For example, if an interface is defined to return a list of items and the list is not guaranteed to be short, then it should not have the entire list returned in one, possibly enormous, set of data. This issue has been addressed primarily through the use of *iterator* objects. When a list of items is to be returned initially, a number of the items are returned along with an object reference to an iterator object. The client may use the iterator object to retrieve the remainder of the list in manageable subsets.

  For complex characteristics of interfaces, we have chosen to represent those characteristics as methods instead of attributes. This not only has scalability advantages, but also performance advantages.
- Certain design decisions were made that effects the programming model that results from the style of interface definition selected. In some cases interfaces have been designed or organized simply so that the programmer suffers the minimal amount of complexity. Two areas where this is particularly true are the organization of the common type definitions and the definitions of exceptions.

3.1 SysAdminTypes Module

This module defines the common types for the management facilities in this specification. Having these types defined in a common IDL module, allows an application developer to simply recall the appropriate language mapping from an IDL source in a consistent manner. If these types were defined in the module which primarily used them, the application developer would have to remember in which module these various types were defined. This is due to the IDL scoping rules that require operation names to have mappings dependent upon the derived interface by which they are inherited, and types to have mappings dependent upon the defining scope.

3.1.1 Specified IDL

```
//
// Component Name: SysAdminTypes.idl
//
// Description:
// The SysAdminTypes file defines types and data
// structures that are used frequently throughout the
// development of system administration applications.
//

#ifndef SYSADMINTYPES_IDL
#define SYSADMINTYPES_IDL

#include <orb.idl>
#include <CosNaming.idl>

module SysAdminTypes {

        typedef sequence <CORBA::InterfaceDef> InterfaceDefList;

        typedef CosNaming::NameComponent LabelType;

        struct ObjectLabel {
                Object objref;
                LabelType label;
        };

        struct LabelExpression {
                string id_regex;
                string kind_regex;
        };

        typedef sequence <ObjectLabel> ObjectLabelList;

        // The Platform structure defines elements for information
        // related to the hardware and operating system of a
        // client machine. The elements represent the following
        // information.

        struct Platform {
                string host_name;
                string machine_hardware_name;
                string operating_system_name;
                string operating_system_version;
                string operating_system_release;
        };
```

```
};

#endif //SYSADMINTYPES_IDL
```

3.2 SysAdminExcept Module

The design of the interfaces and systems management framework in this specification have been influenced by the philosophy of making a few, general exception types defined in a common IDL module, specializing exceptions by use of their data fields, and the integration with an XPG messaging system. Limiting the number and types of exceptions allows future implementations to specialize an exception without the redefinition or subtyping of the original interface. In addition, it allows clients to have simpler, more robust exception handling code by having a string comparison for a smaller number of exception types. The inclusion of XPG message information in an exception allows the server, at the point of error detection, to specify a minimal amount of context and error information for eventual presentation to the user. As the exception is returned, other servers that pass the exception through to other clients may add additional information related to the context or nature of the error. This approach allows an administrator to receive more information about an exception other than simply ''something undesirable happened'', and allows the potential for the administrator to alter the operation intelligently and to hopefully succeed upon retry.

3.2.1 Specified IDL

```
//
// Component Name: SysAdminExcept.idl
//
// Description:
// The SysAdminExcept module defines the exceptions
// commonly used by system management applications.
//

#ifndef SYSADMINEXCEPT_IDL
#define SYSADMINEXCEPT_IDL

#include <SysAdminTypes.idl>

typedef sequence<any> MsgContext;

#define XPG_FIELDS                      \
        string  type_name;              \
        string  catalog;        \
        long    key;                    \
        string  default_msg;            \
        long    time_stamp;             \
        MsgContext      msg_context;

module SysAdminException {
        exception ExException {
                XPG_FIELDS

        };

        exception ExFailed {
                XPG_FIELDS
                string  operation_name;
        };

        exception ExInvalid {
                XPG_FIELDS
                string  resource_name;
        };
```

```
        exception ExNotUniqueLabel {
        XPG_FIELDS
        SysAdminTypes::LabelType        label;
        };

        exception ExNotFound {
                XPG_FIELDS
                string    resource_name;
        };

        exception ExExists {
                XPG_FIELDS
                string    resource_name;
        };

        exception ExObjNotFound {
                XPG_FIELDS
                string    resource_name;
        };
};

#endif //SYSADMINEXCEPT_IDL
```

3.3 Identification Module

The **Identification** module defines interfaces that allow an object to be identified.

3.3.1 Interfaces and Operations

Module	Interface	Operation
Identification	Labeled	*get_label* *set_label*

Table 3-1 Interfaces and Operations for the Identification Module

3.3.2 Specified IDL

```
//
// Component Name: Identification.idl
//
// Description:
// This module defines the methods that implement an object's
// label, which is a name that uniquely identifies the object
// within an environment.
//

#ifndef IDENTIFICATION_IDL
#define IDENTIFICATION_IDL

#include <SysAdminTypes.idl>

module Identification {

        interface Labeled {

                SysAdminTypes::LabelType get_label();

                void set_label (in SysAdminTypes::LabelType label);

        };
};

#endif //IDENTIFICATION_IDL
```

3.3.3 Identification::Labeled Interface

The **Identification::Labeled** interface provides the operations needed to allow an object to have a label. The underlying type of the label is a **CosNaming::NameComponent**, which allows an object's label to also be used as its name within a COS Naming Service name space (see reference **COS Volume 1**).

3.3.3.1 The set_label Operation

The *set_label* operation sets the object's label to the value passed in the label parameter. This operation is only intended to be invoked by an **InstanceManager** at the time an object is created. Conforming implementations are highly discouraged from allowing subsequent invocations upon objects that have already had their labels set. Such invocations may lead to unpredictable behavior in the other services described in this specification.

Syntax

```
void set_label(in SysAdminTypes::LabelType label);
```

Exceptions

CORBA 1.2 standard exceptions.

3.3.3.2 The get_label Operation

The *get_label* operation returns the current value of the objects label.

Syntax

```
SysAdminTypes::LabelType get_label();
```

Exceptions

CORBA 1.2 standard exceptions.

3.4 SysAdminLifeCycle Module

The **SysAdminLifecycle** module defines the **Location** and **HostLocation** interfaces and their operations. These interfaces allow the object that supports them to identify a specific location at which an operation (typically object creation) should be performed. The **Location** interface is extremely generic and supports no attributes or operations. It is mainly intended to be used as an abstract interface that will be subtyped to form interfaces that provide more specific information about particular types of locations. The **HostLocation** interface is an example of a subtype of the **Location** interface that is capable of providing information about a particular type of location: in this case, a host. It is defined due to it's general purpose usefulness, and to provide an example of how the **Location** interface can be used.

The possible definitions for subtypes of the Location interfaces are quite broad and are not specified. Because of this, the *scope* of applicability of the Location interface and its subtypes is guaranteed only within a single vendor's framework implementation.

3.4.1 Interfaces and Operations

Module	Interface	Operation
SysAdminLifeCycle	Location	
	HostLocation *Inheritance:* SysAdminLifeCycle::Location	*get_platform*

Table 3-2 Interfaces and Operations for the SysAdminLifeCycle Module

3.4.2 Specified IDL

```
//
// Component Name: SysAdminLifecycle.idl
//
//
// Description:
// The SysAdminLifeCycle module defines interfaces
// for specifying objects. Currently interfaces for copying and
// moving objects are not supported. These will be added as needed.
//

#ifndef SYSADMINLIFECYCLE_IDL
#define SYSADMINLIFECYCLE_IDL

#include <SysAdminTypes.idl>

module SysAdminLifeCycle {

        // The Location interface allows the specification of
        // the location for lifecycle operations to occur.

        interface Location {
        };

        // the HostLocation interface allows the specification
        // of a particular client machine on which the
        // lifecycle operation should execute
```

```
        interface HostLocation : Location {

                SysAdminTypes::Platform get_platform();

        };

};

#endif // SYSADMINLIFECYCLE_IDL
```

3.4.3 SysAdminLifeCycle::Location Interface

The **SysAdminLifeCycle::Location** interface defines no operations and does not inherit any other interfaces. It's primary purpose is to introduce the idea that, for object creation, the location of where to create an object must somehow be specified. However, since the concept of location is very ORB implementation dependent, how to specify location cannot be addressed in this specification. By introducing this interface, it allows other interfaces (such as **ManagedInstances::BasicInstanceManager**) to specify that they require a Location object in order to perform particular operations.

3.4.4 SysAdminLifeCycle::HostLocation Interface

The **SysAdminLifeCycle::HostLocation** interface defines the *get_platform* operation, which returns information related to the platform on which the object currently resides. This information is related to the hardware type and the operating system.

3.4.4.1 Inherited Interfaces

The **SysAdminLifeCycle::HostLocation** interface inherits from the **SysAdminLifecycle::Location** interface to represent the fact that it can be viewed as a specific type of location.

3.4.4.2 The get_platform Operation

The *get_platform* operation gets the client machine information of the host on which the target object resides. This information should be accurate even after the target object migrates from one client machine to another. The **host_name** element of the **Platform** structure is typically the name by which the client machine is known to the communications network. The **machine_hardware_name** is the standard name that identifies the hardware on which the object is running. The **operating_system_name** element is the current operating system under which the object is running. The **operating_system_release** element specifies the specific release number of the operating system (for example, 5.4). The **operating_system_version** element specifies the exact version of the operating system. Often times a version of an operating system is determined by its relationship to hardware (for example, generic).

Syntax

```
SysAdminTypes::Platform get_platform();
```

Exceptions

CORBA 1.2 standard exceptions.

3.5 ManagedSets Module

The **ManagedSets** module defines interfaces that provide operations for establishing two way reference relationships between objects.

3.5.1 Interfaces and Operations

Module	Interface	Operation
ManagedSets	Set *Inheritance:* ManagedSets::Member	*get_cardinality* *add_object* *add_n_objects* *remove_object* *remove_n_objects* *get_members* *intersection_members* *union_members*
	Member *Inheritance:* Identification::Labeled	*add_backref* *get_backrefs* *remove_backref*
	FilteredSet *Inheritance:* ManagedSets::Set	*find_members* *lookup_object* *lookup_labels*
	SetIterator	*next_one* *next_n* *destroy*
	MemberIterator	*next_one* *next_n* *destroy*
	MemberLabelIterator	*next_one* *next_n* *destroy*

Table 3-3 Interfaces and Operations for the ManagedSets Module

3.5.2 Specified IDL

```
//
// Component Name: ManagedSets.idl
//
// Description:
// The following interfaces provide the functionality
// for support of managed sets.
//

#ifndef MANAGEDSETS_IDL
#define MANAGEDSETS_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <Identification.idl>
```

```
module ManagedSets {

//--------------------------------------------------------------------
//Forward references
//--------------------------------------------------------------------
interface Member;
interface SetIterator;
interface MemberIterator;
interface MemberLabelIterator;
interface Set;
interface FilteredSet;

//--------------------------------------------------------------------
//typedefs needed for the managed set interfaces
//--------------------------------------------------------------------
typedef sequence <Set>              SetList;
typedef sequence <Member>           MemberList;
typedef SysAdminTypes::ObjectLabel     MemberLabel;
typedef SysAdminTypes::ObjectLabelList MemberLabelList;

        interface Member : Identification::Labeled {

                void add_backref (
                        in Set s
                );

                void get_backrefs (
                        in unsigned long how_many,
                        out SetList s_list,
                        out SetIterator iterator
                );

                void remove_backref (
                        in Set s
                ) raises (
                        SysAdminException::ExNotFound
                );

        }; // End Member interface

        interface SetIterator {

                boolean next_one (
                        out Set s
                );

                boolean next_n (
                        in unsigned long how_many,
                        out SetList s_list
                );

                void destroy();

        }; // End SetIterator interface


        interface MemberIterator {

                boolean next_one (
```

```
                out Member m
        );

        boolean next_n (
                in unsigned long how_many,
                out MemberList m_list
        );

        void destroy();

}; // End of MemberIterator

interface ObjectLabelIterator {

        boolean next_one (
                out ObjectLabel ol
        );

        boolean next_n (
                in unsigned long how_many,
                out ObjectLabelList ol_list
        );

        void destroy();

};  // End of ObjectLabelIterator

typedef ObjectLabelIterator MemberLabelIterator;


interface Set : Member {

        unsigned long get_cardinality();

        void add_object (
                in boolean add_backref,
                in Member m
        ) raises (
                SysAdminException::ExNotUniqueLabel
        );

        void add_n_objects (
                in MemberList m_list,
                out MemberList not_added
        );

        void remove_object (
                in boolean remove_backref,
                in Member m
        ) raises (
                SysAdminException::ExNotFound
        );

        void remove_n_objects (
                in boolean remove_backref,
                in MemberList m_list,
                out MemberList not_removed
        );
```

```
                void get_members (
                        in unsigned long how_many,
                        out MemberList m_list,
                        out MemberIterator iterator
                );

                void intersection_members (
                        in unsigned long how_many,
                        in SetList s_list,
                        out MemberList m_list,
                        out MemberIterator iterator
                ) raises (
                        SysAdminException::ExInvalid
                );

                void union_members (
                        in unsigned long how_many,
                        in SetList s_list,
                        out MemberList m_list,
                        out MemberIterator iterator
                ) raises (
                        SysAdminException::ExInvalid
                );

        }; // End of Set interface

        interface FilteredSet : Set {

                void find_members (
                        in SysAdminTypes::InterfaceDefList interfaces,
                        in SysAdminTypes::LabelExpression regular_expression,
                        in unsigned long how_many,
                        out MemberLabelList ml_list,
                        out MemberLabelIterator iterator
                ) raises (
                        SysAdminException::ExNotFound
                );

                Member lookup_object (
                        in SysAdminTypes::LabelType label,
                        in SysAdminTypes::InterfaceDefList interfaces
                ) raises (
                        SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList lookup_labels (
                        in MemberList m_list
                );

        }; // End of FilteredSet interface

}; // End ManagedSets module

#endif //MANAGEDSETS_IDL
```

3.5.3 ManagedSets::Member Interface

The **Member** interface defines the following operations that allow an object to be member of a set object:

- *add_backref*
- *get_backrefs*
- *remove_backref*

3.5.3.1 Inherited Interfaces

The **ManagedSets::Member** interface inherits from the **Identification::Labeled** interface.

3.5.3.2 The add_backref Operation

The *add_backref* operation adds an object reference to an unordered list. The references are to objects supporting the **Set** interface. Adding a back reference provides a means for objects to know about the containment relationships of which they are a part. Typically this operation would never be called by client code, but would be called by an implementation of the **ManagedSets::Set** interface, in particular from the *add_object* or *add_n_objects* operations. Arbitrary use of the *add_backref* operation by client code can, and probably will, result in referential integrity problems in **Set-Member** relationships. It is made a public part of the **Member** interface as it is anticipated that set to set member relationships will exist between objects that exist in different vendors' framework implementations.

Syntax

```
void add_backref(
        in Set s
);
```

Exceptions

CORBA 1.2 standard exceptions.

If the target object already maintains a reference to the input *set s*, the invocation of *add_backref* will be a no-op: no exception will be raised and only one reference to the *set s* will be maintained by the target object.

3.5.3.3 The get_backrefs Operation

The *get_backrefs* operation returns the object references (back references) for the Sets of which the object is a member. If *how_many* is greater than or equal to the total number of back references maintained by the target object, then all of the back references will be returned in *s_list* and *iterator* will be returned as OBJECT_NIL.

Syntax

```
void get_backrefs(
        in unsigned long how_many,
        out SetList s_list,
        out SetIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.3.4 The remove_backref Operation

The *remove_backref* operation removes the back reference to the Set specified by the *s* argument. Typically this operation would never be called by client code, but would be called by an implementation of the **ManagedSets::Set** interface, in particular from the *remove_object* or *remove_n_objects* operations. Arbitrary use of the *remove_backref* operation by client code can, and probably will, result in referential integrity problems in **Set-Member** relationships. It is made a public part of the **Member** interface as it is anticipated that set to set member relationships will exist between objects that exist in different vendors' framework implementations.

Objects that inherit from both **ManagedSets::Set** and **CosLifeCycle::LifeCycleObject** will inherit the latter's remove operation. If an object which is a Set destroys itself via the remove operation, then it must first remove itself from its member objects' lists of backrefs by invoking *remove_backref* upon each of them in turn, with the Set's own object reference as the input parameter.

Syntax

```
void remove_backref(
        in Set s
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

If the object upon which this operation is invoked is not a member of the set indicated by the input argument the **SysAdminException::ExNotFound** exception is raised.

3.5.4 ManagedSets::SetIterator Interface

The **SetIterator** interface defines the following operations that allow an object to iteratively request a list of sets to which an object is a member. The **SetIterator** provides a static view of the sets to which the object belonged at the time the iterator was created, and does not reflect subsequent changes to the object's membership in sets. The operations provided are:

- *next_one*
- *next_n*
- *destroy*

3.5.4.1 The next_one Operation

The *next_one* operation returns the next object. If the last set object is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_one(
        out Set s
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.4.2 The next_n Operation

The *next_n* operation returns at most the requested number of set objects. If the last set object is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_n(
        in unsigned long how_many,
        out SetList s_list
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.4.3 The destroy Operation

The *destroy* operation destroys the iterator.

Syntax

```
void destroy();
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.5 ManagedSets::MemberIterator Interface

The **MemberIterator** interface defines the following operation that allows an object to iteratively request a list of members that belong to a set. The **MemberIterator** provides a static view of the set's members at the time the iterator was created, and does not reflect subsequent changes to the set's membership. The operations provided are:

- *next_one*
- *next_n*
- *destroy*

3.5.5.1 The next_one Operation

The *next_one* operation returns the next set member object. If the last set member object is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_one(
        out Member m
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.5.2 The next_n Operation

The *next_n* operation returns at most the requested number of set member objects. If the last set member object is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_n(
        in unsigned long how_many,
        out MemberList m_list
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.5.3 The destroy Operation

The *destroy* operation destroys the iterator.

Syntax

```
void destroy();
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.6 ManagedSets::ObjectLabelIterator Interface

The ObjectLabelIterator interface is a general-purpose iterator interface that defines the following operations that allow an object to iteratively request a list of **ObjectLabel** structures that correspond to objects that have a special relationship to the target object whose ‘‘get’’ or ‘‘list’’ operation generated the iterator. In the case of **ManagedSets**, this means that the **ObjectLabels** correspond to objects that are members of the target set. In the case of **PolicyRegions**, the **ObjectLabels** correspond to policy-driven objects that are supported by the target **PolicyRegion** (see, for example, Section 3.7.5.4 on page 74). The **ObjectLabelIterator** provides a static view of the objects that had this special relationship to the original target object at the time the iterator was created, and does not reflect subsequent changes to the list of objects that continue to have this special relationship. The operations provided are:

- *next_one*
- *next_n*
- *destroy*

3.5.6.1 The next_one Operation

The *next_one* operation returns the next **ObjectLabel** structure. If the last **ObjectLabel** structure is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_one (
        out MemberLabel ol;
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.6.2 The next_n Operation

The *next_n* operation returns at most the requested number of **ObjectLabel** structures. If the last **ObjectLabel** structure is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed. If how_many is greater than or equal to the total number of **ObjectLabel** structures remaining in the iterator object, then all of the **ObjectLabel** structures will be returned in *ol_list* and no exception will be raised.

Syntax

```
boolean next_n(
        in unsigned long how_many,
        out ObjectLabelList ol_list
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.6.3 The destroy Operation

The *destroy* operation destroys the iterator.

Syntax

```
void destroy();
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.7 ManagedSets::Set Interface

The set relationship supported by a set object is a many-to-many reference relationship. An object may be a member of an arbitrary number of sets. Sets are a way for system administrator to group objects and to develop subsets of objects and resources to be managed similarly. In addition to the Containment functionality of the basic set, behaviors are supported to provide a means of selecting a subset of the set's membership.

The **Set** interface defines the following operations that provide basic set object functionality:

- *get_cardinality*
- *add_object*
- *add_n_objects*
- *remove_object*
- *remove_n_objects*
- *get_members*
- *intersection_members*
- *union_members*

3.5.7.1 Inherited Interfaces

The **ManagedSets::Set** interface inherits from the **ManagedSets::Member** interface, thus allowing sets to be arranged into hierarchies.

3.5.7.2 The get_cardinality Operation

The *get_cardinality* operation returns the current number of members of the set.

Syntax

```
unsigned long get_cardinality();
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.7.3 The add_object Operation

The *add_object* operation adds a new object reference to the set of member objects.

If the *add_backref* parameter is TRUE, then the member object being added is notified to add a back reference to the set. If the *add_backref* parameter is FALSE, then the newly-contained member object is not notified. This is useful for the case in which a member object adds itself to the set; the object does not need the notification to add a back reference (it already knows it is being added).

Syntax

```
void add_object(
        in boolean add_backref,
        in Member m
) raises (
        SysAdminException::ExNotUniqueLabel
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotUniqueLabel**.

If the label of the member being added to the set is not unique among members of the set, the **SysAdminException::ExNotUniqueLabel** exception may be raised subject to the following conditions:

1. If the input object **m** has the same label as an already-existing set member and if the **CORBA::Object** defined *is_equivalent* method returns TRUE when invoked on the input object **m** passing the already existing set member, no exception is raised and there remains only one reference to that object from the set.
2. If the input object **m** has the same label as an already-existing set member and if the **CORBA::Object** defined *is_equivalent* method returns FALSE when invoked on the input object **m** passing the already existing set member, then the **SysAdminException::ExNotUniqueLabel** exception is raised. The client is thus alerted that it is holding a superfluous reference to the set member which it should discard in favor of the reference that the set already holds.

Note that a label is comprised of two components: an *id* field and a *kind* field. The label of the input object **m** is only considered to be the same as that of an already-existing member if both of these fields are identical to that of the member.

3.5.7.4 The add_n_objects Operation

The *add_n_objects* operation adds new object references to the set of **Member** object *held* by a Set. If any of the objects were unable to be added to the Set, their object references are returned for appropriate error handling. If some objects are not able to be added, the remainder of the objects are added. Note that the most likely reason why an object was not added is because its label is identical to that of an object that is already a member of the set (see Section 3.5.7.3 on page 43).

Syntax

```
void add_n_objects(
        in MemberList m_list,
        out MemberList not_added
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.7.5 The remove_object Operation

The *remove_object* operation removes the member object specified by the *m* argument from a Set object. The *remove_object* operation does not remove the specified objects. It only removes references to the objects.

If the *remove_backref* argument is set to TRUE, then the member object being removed is notified to remove its back reference to the set. If the *remove_backref* argument is FALSE, the contained object is not notified. This operation is useful when a contained object is removed. Upon deletion, it removes itself from all Sets. There is no need for the set object to notify the member object to remove a back reference in this scenario.

Syntax

```
void remove_object(
        in boolean remove_backref,
        in Member m
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

If the object indicated by the second input parameter is not a member of the target set, the **SysAdminException::ExNotFound** exception is raised.

3.5.7.6 The remove_n_objects Operation

The *remove_n_objects* operation removes the member objects, specified by the *s_list* argument, from a Set object. The *remove_n_objects* operation does not remove the specified objects. It only removes references to the objects. If the *remove_backref* argument is set to TRUE, then the member object being removed is notified to remove its back reference to the Set. If the *remove_backref* argument is FALSE, the member object is not notified. This operation is useful when member objects are to be destroyed. Upon deletion they remove themselves from all Sets. There is no need for the Set object to notify the member objects to remove a back reference in this scenario.

Syntax

```
void remove_n_objects(
        in boolean remove_backref,
        in MemberList m_list,
        out MemberList not_removed
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.7.7 The get_members Operation

The *get_members* operation returns the object references of object's members. If *how_many* is greater than or equal to the total number of members In the Set, then all of the members will be returned in *m_list* and *iterator* will be returned as OBJECT_NIL.

Syntax

```
void get_members (
        in unsigned long how_many,
        out MemberList m_list,
        out MemberIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.5.7.8 The intersection_members Operation

The *intersection_members* operation provides the list of members that are the intersection of sets. Given a list of sets, a list of members will be returned that represents the intersection of the target set and all of the sets specified in the *s_list* argument. The initial size of *m_list* is determined by the *how_many* argument. The rest of the **Member** objects, if any, can be retrieved using the returned iterator object. When *how_many* is equal to or greater than the total number of members that could potentially be returned, all of the members will be returned in *s_list* and the iterator will be returned as OBJECT_NIL.

Syntax

```
void intersection_members (
        in unsigned long how_many,
        in SetList s_list,
        out MemberList m_list,
        out MemberIterator iterator
) raises (
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExInvalid**.

If *s_list* is an empty list or if an element of *s_list* is OBJECT_NIL, the **SysAdminException::ExInvalid** exception is raised. This is not the same thing as taking the intersection of a Set with the empty set (a Set that contains no Members), which is allowed and which will not cause this exception to be raised.

3.5.7.9 The union_members Operation

The *union_members* operation provides the list of members that represent the union of the sets. Given a list of sets, a list of members will be returned that represents the union of the target set and all of the sets specified in the *s_list* argument. This operation ensures no duplicates in the member list. The initial size of *m_list* is determined by the *how_many* argument. The rest of the **Member** objects, if any, can be retrieved using the returned iterator object. When *how_many* is equal to or greater than the total number of members that could potentially be returned, all of the members will be returned in *s_list* and the *iterator* will be returned as OBJECT_NIL.

Syntax

```
void union_members (
        in unsigned long how_many,
        in SetList s_list,
        out MemberList m_list,
        out MemberIterator iterator
) raises (
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExInvalid**.

If *s_list* is an empty list or if an element of *s_list* is OBJECT_NIL, the **SysAdminException::ExInvalid** exception is raised. This is not the same thing as taking the union of a Set with the empty set (a Set that contains no Members), which is allowed and which will not cause this exception to be raised.

3.5.8 ManagedSets::FilteredSet Interface

An object supporting the **FilteredSet** interface is called a filtered set. A filtered set is able to filter across the the member objects and return a subset of its membership based on simple search criteria, which supports filtering based on the interfaces supported by members, or the labels of members.

The **FilteredSet** interface defines the following operations that enable the management of managed sets:

- *find_members*
- *lookup_object*
- *lookup_labels*

3.5.8.1 Inherited Interfaces

The **ManagedSets::FilteredSet** interface inherits from the **ManagedSets::Set** interface:

3.5.8.2 The find_members Operation

The *find_members* operation returns a list of the object references for the objects contained in a set meeting the specified selection criteria. Objects may be selected on the basis of the interfaces they support and their labels. The interfaces argument is a sequence of types defined in the **InterfaceDef** interface. Only objects that support all of the interfaces in the sequence are returned. If this sequence is of zero length, then objects may be returned regardless of their supported interfaces. The *regular_expression* argument specifies a POSIX 1003.2 regular expression for matching on the basis of object labels. Only objects that have labels matching the regular expression are returned. If a regular expression is not specified, then objects will be selected irrespective of their labels. Note that labels are comprised of two components: an *id* field and a *kind* field. A client can specify a regular expression to be matched for one, both, or neither of these fields.

Both the object references and the labels of the objects are returned. The order of the objects returned is unspecified. If a member of the generic set does not have a label, then it will not match any regular expressions.

The initial size of *ml_list* is determined by the *how_many* argument. The rest of the **MemberLabel** structures, if any, can be retrieved using the returned iterator object. If *how_many* is greater than or equal to the total number of members that match the filter criteria, then all of the members will be returned in *ml_list* and iterator will be returned as OBJECT_NIL.

Syntax

```
void find_members (
        in SysAdminTypes::InterfaceDefList interfaces,
        in SysAdminTypes::LabelExpression regular_expression,
        in unsigned long how_many,
        out MemberLabelList ol_list,
        out MemberLabelIterator iterator
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised if no objects in the set meet the specified selection criteria.

3.5.8.3 The lookup_object Operation

The *lookup_object* operation returns an object reference to the set member that has the same label as the label specified in the *label* argument. Labels are unique within sets, therefore at most one object with that label can be found and returned. However, if the *kind* field of the *label* argument is specified as a zero length string, only the *id* field of the label is used in searching for the set member to return. Since the *id* fields of a set's members are not necessarily unique within the set, this operation will return one of possibly many set members that have the matching *id* field.

The *interfaces* argument specifies a sequence of interfaces that the object with the matching label must support in order to be returned. If a zero length sequence of interface definitions is passed, any object with a matching label will be returned.

Syntax

```
Member lookup_object(
        in SysAdminTypes::LabelType label,
        in SysAdminTypes::InterfaceDefList interfaces
)raises(
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised when no object in the set has the specified label.

3.5.8.4 The lookup_labels Operation

The *lookup_labels* operation returns a list of object references and their labels. The objects whose labels are requested are specified by the *m_list* argument. The objects and their labels are returned in the same order in which they were input in *m_list*. If *m_list* contains duplicate members, then the returned **ObjectLabelList** will contain the corresponding duplicate object labels in list positions that correspond to the input duplicate members' list positions. If the input *m_list* contains elements that are set to *Object::_nil()*, then the returned **ObjectLabelList** will contain **ObjectLabels** whose objref members are *Object::_nil()*. Similarly, input objects that are not members of the set cause a *Object::_nil()* to be placed in the output list in place of the errant input object. Finally, errors in processing a list element (for example, an exception when getting the object's label) cause a *Object::_nil()* to be placed in the output list in place of the errant object. These ''null'' **ObjectLabels** will appear in the output list in list positions that correspond to the appropriate input members' list positions, and the *kind* field of the label will be used to explain why the object field contains a *Object::_nil()* value. If *m_list* is zero-length, then the output **ObjectLabelList** will also be zero-length.

Syntax

```
SysAdminTypes::ObjectLabelList lookup_labels(
        in MemberList m_list
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.6 ManagedInstances Module

The **ManagedInstances** module defines the **Instance**, **BasicInstanceManager**, **InstanceManager** and **Library** interfaces and their operations. The **ManagedInstances::Instance** interface defines operations that determine and report the instance manager of an object instance. The **ManagedInstances::BasicInstanceManager** interface defines operations that encapsulate type-specific information related to object creation and instances of the object type. The **ManagedInstances::InstanceManager** extends this to also include information related to policy for instances of the object type. The **ManagedInstances::Library** interface defines operations for the creation and finding of instance manager objects.

3.6.1 Interfaces and Operations

Module	Interface	Operation
ManagedInstances	Instance *Inheritance* CosLifeCycle::LifeCycleObject ManagedSets::Member	*get_manager* *get_type_name* *get_resource_location*
	BasicInstanceManager *Inheritance* CosLifeCycle::LifeCycleObject CosLifeCycle::GenericFactory ManagedSets::FilteredSet	*get_instances_interface*
	InstanceManager *Inheritance* ManagedInstances::BasicInstanceManager Policies::PolicyObjectAdmin	
	Library *Inheritance:* CosLifeCycle::FactoryFinder CosLifeCycle::GenericFactory ManagedSets::FilteredSet	
	PolicyRegionsInstanceManager *Inheritance:* ManagedInstances::InstanceManager	

Table 3-4 Interfaces and Operations for the Instances Module

3.6.2 Specified IDL

```
//
// Component Name: ManagedInstances.idl
//
// Description:
// The following interfaces provide the functionality
// for support of instances and instance managers.
//

#ifndef MANAGEDINSTANCES_IDL
#define MANAGEDINSTANCES_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <SysAdminLifeCycle.idl>
#include <CosLifeCycle.idl>
#include <Policies.idl>
#include <ManagedSets.idl>


module ManagedInstances {

        // Forward reference
        interface Instance;
        interface BasicInstanceManager;
        interface InstanceManager;
        interface Library;

        interface Instance :
                CosLifeCycle::LifeCycleObject,
                ManagedSets::Member
        {

                BasicInstanceManager get_manager();

                string get_type_name();

                SysAdminLifeCycle::Location get_resource_location();

        }; // End of Instance interface

        interface Library :
                CosLifeCycle::FactoryFinder,
                CosLifeCycle::GenericFactory,
                ManagedSets::FilteredSet
        {
        }; // End of Library interface

        interface BasicInstanceManager :
                CosLifeCycle::LifeCycleObject,
                CosLifeCycle::GenericFactory,
                ManagedSets::FilteredSet
        {

                CORBA::InterfaceDef get_instances_interface();

        }; // End of BasicInstanceManager interface
```

```
        interface InstanceManager :
                BasicInstanceManager,
                Policies::PolicyObjectAdmin

        {
        }; // End of InstanceManager interface

        interface PolicyRegionsInstanceManager :
                InstanceManager
        {
        }; // End of PolicyRegionsInstanceManager interface

};  // End ManagedInstances module

#endif // MANAGEDINSTANCES_IDL
```

3.6.3 ManagedInstances::Instance Interface

The **ManagedInstances::Instance** interface defines the operations required for an object to be managed by an instance manager. An instance is able to report the IDL interface name for its principal interface type, and the object reference of the instance manager associated with the type.

The **ManagedInstance::Instance** interface defines the following operations:

- *get_manager*
- *get_type_name*
- *get_resource_location*

3.6.3.1 Inherited Interfaces

The **Instance** interface inherits from the following interfaces:

- **CosLifeCycle::LifeCycleObject**
- **ManagedSets::Member**

3.6.3.2 Unique Behavior of Inherited Interfaces

Operations defined within these interfaces may have unique behavior when inherited by the Instance interface.

CosLifeCycle::LifeCycleObject

The *remove* operation must invoke the *ManagedSets::Set::remove_object* operation with it's own object reference specified for removal on all sets that contain the target object at the time of deletion. This operation notifies the sets that one of it's member objects is being removed. This also applies to the instance's membership in its instance manager. The only way that an instance may be removed from its instance manager is by virtue of a client having invoked the instance's inherited *remove* operation to destroy the instance.

3.6.3.3 The get_manager Operation

The *get_manager* operation returns the object reference of the instance manager that manages the target object.

Syntax

```
BasicInstanceManager get_manager( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.6.3.4 The get_type_name Operation

The *get_type_name* operation returns the fully scoped IDL interface name of an instance.

Syntax

```
string get_type_name( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.6.3.5 The get_resource_location Operation

The *get_resource_location* operation returns the object reference of the **SysAdminLifeCycle::Location** for the location where the current object resides. This operation can be used during life cycle-defined operations. For example, when the location of one object needs to be determined so that another object may be created in the same location, the *get_resource_location* operation is invoked. The returned object reference is used as an input to the *CosLifeCycle::GenericFactory::create_object* operation.

Syntax

```
SysAdminLifeCycle::Location get_resource_location( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.6.4 ManagedInstances::BasicInstanceManager Interface

The **ManagedInstances::BasicInstanceManager** interface defines operations needed to group objects into sets for purposes of object management. An object supporting the **ManagedInstances::BasicInstanceManager** interface manages a group of objects of one type. Basic instance managers maintain references to all instances they manage, and are able to report the interface that is supported by instances they manage. Additionally, instance managers are able to create instances of the type of objects they support. Instances created and managed by a **BasicInstanceManager** must support the **ManagedInstances::Instance** interface.

Basic instance managers are created by invoking the *create_object* operation on a library object that is responsible for supporting object types within a specified scope of influence, which may be some subset of the overall managed environment.

The **ManagedInstances::BasicInstanceManager** interface defines the following operations:

- *get_instances_interface*

3.6.4.1 Inherited Interfaces

The **BasicInstanceManager** interface inherits from the following interfaces:

- **CosLifeCycle::LifeCycleObject**
- **CosLifeCycle::GenericFactory**
- **ManagedSets::FilteredSet**

3.6.4.2 Unique Behavior of Inherited Interfaces

Operations defined within these interfaces may have unique behavior when inherited by the **BasicInstanceManager** interface:

CosLifeCycle::LifeCycleObject

BasicInstanceManager inherits the remove operation. Conforming implementations of the **BasicInstanceManager** must disallow the invocation of the remove operation while instances are presently contained by the instance manager. Further, as with instances, an instance manager may only be removed from the library that contains it as a consequence of a client having invoked the remove operation upon the instance manager in order to destroy it.

CosLifeCycle::GenericFactory

BasicInstanceManager uses this interface to create objects of the type it manages.

The *k* argument is used with both the *create_object* and supports operations. It is defined to take the following:

kind **field**	*id* **field**
‘‘object interface’’	The fully scoped name of the principal IDL interface for the type of instance to be created.
‘‘object implementation’’	The name of the implementation to be used with the instance to be created.

For the *supports* operation the *k* argument is required. It is implementation dependent as to whether just ‘‘object interface’’ or ‘‘object implementation’’, or both, are required to be specified. If the **BasicInstanceManager** is capable of creating objects of the specified interface and implementation, it returns TRUE, otherwise it returns FALSE.

For the *create_object* operation, the *k* argument is optional and may be specified as an empty sequence (that is, sequence of length zero). This is because the basic instance manager already knows the type of object it creates, including the **InterfaceDef** and **ImplementationDef** for that type. If the *k* argument is specified, the values specified must be consistent with the **InterfaceDef** and **ImplementationDef** associated with this basic instance manager object. If it is not, the **CosLifeCycle::NoFactory** exception is raised.

For the *create_object* operation, after the new instance is created, the *add_object* operation on the basic instance manager must be invoked to add the new instance to this basic instance manager’s set.

The following table defines the name/value pairs that are passed in the *the_criteria* argument of the *create_object* call.

Name	Type	Value Interpretation
''labelid''	string	A symbolic name for the instance being created. The *id* field of the instance's label will be set to this value, and the *kind* field will be set to the concatenation of the Instance Manager's *id* and *kind* fields (separated by a vertical bar, ''|''). Note that the latter indicates the name of the Library object that created the InstanceManager.
''location''	Location	Object reference of an object that indicates the location where the new instance should be created.

If ''labelid'' or ''location'' are not specified in *the_criteria* argument, or if the string passed in the ''labelid'' matches the id field of the label of any existing members of this instance manager, the **CosLifeCycle::InvalidCriteria** exception will be raised.

Note that an instance's label is the primary distinguishing characteristic for that instance. For that reason, an instance's label must be unique in space and time. The ''labelid'' criterion above must be constructed with that end in view.

ManagedSets::FilteredSet

BasicInstanceManager inherits the *add_object*, *add_n_objects*, *remove_object* and *remove_n_objects* operations, among others, via its inheritance from **ManagedSets::FilteredSet**. Since instances are intended to spend their entire existence as members of the instance manager that created them, these four operations must be constrained to work only in the context of other operations that affect an instance's lifecycle. Specifically, the instance manager's inherited *remove_object* operation should only be invoked by an instance's inherited **CosLifeCycle::LifeCycleObject::remove** operation, and *add_object*, *add_n_objects* and *remove_n_objects* must be disabled.

3.6.4.3 The get_instances_interface Operation

The *get_instances_interface* operation returns an object reference to a description of the interface supported by instances supported by this basic instance manager.

Syntax

```
CORBA::InterfaceDef get_instances_interface();
```

Exceptions

CORBA 1.2 standard exceptions.

3.6.5 ManagedInstances::InstanceManager Interface

The **ManagedInstances::InstanceManager** interface provides all of the **ManagedInstances::BasicInstanceManager** capabilities, and in addition provides capabilities that support the specification of policy to be associated with managed instances. This is done through the inheritance of the **Policies::PolicyObjectAdmin** interface. Instances created and managed by an **InstanceManager** must support the **PolicyRegions::PolicyDrivenBase** interface.

Instance managers are created by invoking the *create_object* operation on a library object that is responsible for supporting object types within a specified scope of influence, which may be some subset of the overall managed environment.

The **ManagedInstances::InstanceManager** interface does not introduce any new operations other than those that are inherited.

3.6.5.1 Inherited Interfaces

The **InstanceManager** interface inherits from the following interfaces:

- **ManagedInstances::BasicInstanceManager**
- **Policies::PolicyObjectAdmin**

3.6.5.2 Unique Behavior of Inherited Interfaces

Operations defined within these interfaces may have unique behavior when inherited by the **InstanceManager** interface.

CosLifeCycle::GenericFactory

InstanceManager objects have the same unique behaviors for this inherited interface as those described for **BasicInstanceManager** objects.

For the *create_object* operation, after the new instance is created and added to this instance manager, the *add_object* operation on the policy region must be invoked to cause the created policy driven base object to be added to the policy region.

The following table defines the name/value pairs that are passed in the *the_criteria* argument of the *create_object* call in addition to those defined for **BasicInstanceManager**

Name	Type	Value Interpretation
‘‘policy regions’’	PolicyRegionList	A list of object references of the **PolicyRegion** objects that the new **PolicyDrivenBase** object should be added to. At least one PolicyRegion must be in the list.
‘‘nested initialization’’	boolean	An indicator if initialization policy from nested policy regions should be used. TRUE indicates that it should, while FALSE indicates that it should not.

If ‘‘policy regions’’ is not specified in *the_criteria* argument, the **CosLifeCycle::InvalidCriteria** exception is raised. If an error occurs during the processing of the ‘‘policy regions’’ criterion, then the **CosLifeCycle::CannotMeetCriteria** exception is raised and all processing that has so far occurred in the **create_object**( ) operation is rolled back as if it was a transaction, including the deletion of the newly-created instance.

If ‘‘nested initialization’’ is not specified, then nested initialization will not take place — the default behavior will be the same as if a value of FALSE had been specified. The ordering of initialization policy objects on which the *initialize_policy_driven_object* operation will be called is defined as follows:

- The order will be the same as the ordering of policy regions within the ‘‘policy regions’’ name value pair of the *the_criteria* parameter.
- If nested initialization is to take place, the topmost containing policy region’s initialization policy object is used prior to that of the policy region in the parameter[2].

3.6.6 ManagedInstances::Library Interface

The **ManagedInstances::Library** interface acts as both a factory for the creation of instance managers and as a factory finder for locating instance managers. This includes all of the various instance manager types defined in this specification, namely:

- **ManagedInstances::BasicInstanceManager**
- **ManagedInstances::InstanceManager**

The library also acts as a factory and factory finder for subtypes of these interfaces.

There may be one or more objects that supports the **ManagedInstances::Library** interface in an environment.

The **ManagedInstances::Library** interface does not introduce any new operations other than those that are inherited.

Objects inheriting from the **ManagedInstances::Library** interface may not be destroyed while instance managers are members of those objects.

3.6.6.1 Inherited Interfaces

The **ManagedInstances::Library** interface inherits from the following interfaces:

- **ManagedSets::FilteredSet**
- **CosLifeCycle::GenericFactory**
- **CosLifeCycle::FactoryFinder**

2. It is possible for a policy region itself to be a member of multiple policy regions. When this occurs, the order of initialization policies used is not defined. Because of this, it is recommended that policy regions should not belong to more than one other policy region if deterministic ordering of initialization policies is required.

3.6.6.2 Unique Behavior of Inherited Interfaces

Operations defined within these interfaces may have unique behavior when inherited by the **Library** interface:

ManagedSets::FilteredSet

Library inherits the *add_object*, *add_n_objects*, *remove_object* and *remove_n_objects* operations, among others, via its inheritance from **ManagedSets::FilteredSet**. Since instance managers are intended to spend their entire existence as members of the library that created them, these four operations must be constrained to work only in the context of other operations that affect an instance manager's lifecycle. Specifically, *remove_object* may only be invoked by or as a consequence of an instance manager's inherited **CosLifeCycle::LifeCycleObject::remove** operation, and *add_object*, *add_n_objects* and *remove_n_objects* must be disabled.

CosLifeCycle::GenericFactory

The **Library** uses this interface to create objects that are instances of the **BasicInstanceManager** interface as well as objects that are sub-types of **BasicInstanceManager** (including **InstanceManager**, and **PolicyRegionsInstanceManager**).

The *k* argument is used with both the *create_object* and *supports* operations. It is defined to take the following:

kind **field**	*id* **field**
''object interface''	The fully scoped name of the principal IDL interface for the type of instance manager to be created.
''object implementation''	The name of the implementation to be used with the instance manager to be created.

For both the *supports* and *create_object* operations the *k* argument is required. It is implementation dependent as to whether just ''object interface'' or ''object implementation'', or both, are required to be specified. The ''object interface'' must specify a principal IDL interface that supports the **BasicInstanceManager** interface. For *supports*, if the **Library** is capable of creating an instance manager of the specified type and implementation, it returns TRUE, otherwise it returns FALSE. For *create_object*, if the **Library** is not capable of creating an instance manager of the specified type and implementation, the **CosLifeCycle::NoFactory** exception is raised.

The *the_criteria* argument of the *create_object* operation may pass the elements defined in the following table:

Name	**Type**	**Value Interpretation**
‘‘labelid’’	string	A symbolic name for the instance manager being created. This value will be stored as the *id* field of the instance manager’s label. The *kind* field of the instance manager’s label will be set to the *id* field of the library object upon which this operation was invoked. The *id* and *kind* fields of an instance manager’s label are then concatenated, with a vertical bar (‘‘|’’) between them as a delimiter, to form the *kind* field of any instance that the instance manager creates.
‘‘default initialization’’	InitializationPolicy	Object reference of the default initialization policy object associated with the instance manager being created.
‘‘initializations’’	sequence <InitializationPolicy>	Object references of additional initialization policy objects associated with the instance manager being created.
‘‘default validation’’	ValidationPolicy	Object reference of the default validation policy object associated with the instance manager being created.
‘‘validations’’	sequence <ValidationPolicy>	Object references of additional validation policy objects associated with the instance manager being created.
‘‘location’’	Location	Object reference indicating location of where the instance manager being created should reside.
‘‘type’’	string	The fully scoped name of the principal IDL interface of the type of instances this instance manager will create.
‘‘interface’’	CORBA::InterfaceDef	The InterfaceDef associated with the type of instances this instance manager will create.
‘‘implementation’’	ImplementationDef	The ImplementationDef associated with the type of instances this instance manager will create.

Note that not all criteria are required to be supported in all implementations. It’s important that the client supply sufficient information to describe the type of object the instance manager will create. In some implementations, the *type* criteria alone may be sufficient for this. Other implementations may require the *interface* and *implementation* criteria, but not the *type.* Within a particular implementation, if sufficient information is not supplied by the client to describe the types of objects the Instance Manager will create, the **CosLifeCycle::InvalidCriteria** exception is raised.

If the string passed in the ‘‘labelid’’ matches the id field of the label of any existing members of this library, the **CosLifeCycle::InvalidCriteria** exception is raised.

If an error occurs during the processing of the ‘‘initializations’’ or ‘‘validations’’ criteria, then the **CosLifeCycle::CannotMeetCriteria** exception is raised and all processing that has so far occurred in the **create_object**( ) operation will be rolled back as if it was a transaction, including the deletion of the newly-created instance manager.

CosLifeCycle::FactoryFinder

The *find_factories* operation is invoked on instances of the **ManagedInstances::Library** interface in order to obtain a list of instance managers that meet a set of selection constraints. These constraints are passed in the *factory_key* argument, and are composed as a sequence of **CosNaming::NameComponent** structures, each containing a *kind* field and an *id* field. The following table defines what can be passed in this argument:

kind **field**	*id* **field**
''object interface''	Fully scoped interface name for the instances this instance manager creates
''object implementation''	Name of the implementation for the instances this instance manager creates
''initialization policy''	One of ''with'', ''without''
''validation policy''	One of ''with'', ''without''
''factory interface''	Fully scoped interface name for the instance manager

Note that not all of these selection constraints are required to be supported by all implementations. If the **Library** does not contain any instance manager objects that meet the specified constraints, or if the *factory_key* argument is specified as an empty sequence, the **CosLifeCycle::NoFactory** exception is raised.

3.6.7 ManagedInstances::PolicyRegionsInstanceManager Interface

The **ManagedInstances::PolicyRegionsInstanceManager** interface defines a specialized instance manager that is used to create and manage instances of the **PolicyRegions::PolicyRegion** interface. It enables a policy region to be initialized when it is created, setting it up similar to another policy region specified at object creation time.

3.6.7.1 Inherited Interfaces

The **PolicyRegionsInstanceManager** interface inherits from the **ManagedInstances::InstanceManager** interface.

3.6.7.2 Unique Behavior of Inherited Interfaces

Operations within the following interface have unique behavior when inherited by the **PolicyRegionsInstanceManager** interface:

CosLifeCycle::GenericFactory

The *create_object* operation is invoked on instances of **ManagedInstances::PolicyRegionsInstanceManager** in order to create new policy regions. This operation is the same as is defined for the **ManagedInstances::InstanceManager** interface, with the exception that *the_criteria* argument can also contain the following element:

Name	Type	Value Interpretation
‘‘model policy region’’	PolicyRegion	Object reference to the PolicyRegion that the new PolicyRegion should initially be modeled after. This means that the new policy region will be initialized to support all of the same instance managers (using the same InitializationPolicy and ValidationPolicy objects) as the model policy region.

3.7 PolicyRegions Module

The **PolicyRegions** module defines the **PolicyRegion**, and **PolicyDrivenBase** interfaces and their operations. These policy-related interfaces allow administrators to organize applications to reflect site-specific rules of system administration.

3.7.1 Interfaces and Operations

Module	Interface	Operation
PolicyRegions	PolicyRegion *Inheritance:* PolicyRegions::PolicyDrivenBase ManagedSets::FilteredSet CosLifeCycle::GenericFactory	*add_instance_manager* *remove_instance_manager* *get_instance_manager_list* *set_initialization_policy* *get_initialization_policy* *set_validation_policy* *get_validation_policy* *policy_validation* *is_validation_enabled* *verify_policy* *get_policy_failures* *get_all_initialization_policies* *get_all_enabled_validation_policies*
	PolicyDrivenBase *Inheritance:* ManagedInstances::Instance	*get_policy_region_info* *move_to_policy_region* *add_to_policy_region* *remove_from_policy_region* *list_enabled_validation_policies* *list_initialization_policies*

Table 3-5 Interfaces and Operations for the PolicyRegions Module

3.7.2 Specified IDL

```
//
// Component Name: PolicyRegions.idl
//
// Description:
// The following interfaces provide the functionality
// for support of policy regions.
//

#ifndef POLICYREGIONS_IDL
#define POLICYREGIONS_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <SysAdminLifeCycle.idl>
#include <CosLifeCycle.idl>
#include <ManagedSets.idl>
#include <ManagedInstances.idl>

module PolicyRegions {

// forward references
```

```
interface PolicyRegion;
interface PolicyDrivenBase;

struct PolicyResult {
        PolicyDrivenBase object_verified;
        PolicyRegion containing_region;
        Policies::ValidationPolicy validation_object_used;
        boolean passed_policy;
        string description;
};

typedef sequence<PolicyResult> PolicyResultList;

enum SelectionCriteria {
        all,
        with_initialization,
        with_validation,
        with_validation_enabled,
        with_initialization_or_validation,
        with_initialization_or_validation_enabled
};

typedef sequence<PolicyRegion> PolicyRegionList;

interface PolicyResultIterator {
        boolean next_one (
                        out PolicyResult pr;
        };

        boolean next_n (
                in unsigned long how_many,
                out PolicyResultList pr_list;
        };

        void destroy();

}; // End PolicyResultIterator interface

interface PolicyDrivenBase : ManagedInstances::Instance {

        SysAdminTypes::ObjectLabelList get_policy_region_info();

        void move_to_policy_region (
                in PolicyRegions::PolicyRegion pr_from,
                in PolicyRegions::PolicyRegion pr_to
        ) raises (
                SysAdminException::ExObjNotFound,
                SysAdminException::ExNotFound,
                SysAdminException::ExInvalid
        );

        void add_to_policy_region (
                in PolicyRegions::PolicyRegion pr
        ) raises (
                SysAdminException::ExNotFound,
                SysAdminException::ExInvalid
        );

        void remove_from_policy_region (
```

```
                    in PolicyRegions::PolicyRegion pr
                ) raises (
                    SysAdminException::ExObjNotFound,
                    SysAdminException::ExInvalid
                );

                SysAdminTypes::ObjectLabelList list_enabled_validation_policies (
                    in PolicyRegionList policy_regions,
                    in CORBA::InterfaceDef interface_def,
                    in boolean include_nested
                ) raises (
                    SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList list_initialization_policies
(
                    in PolicyRegionList policy_regions,
                    in CORBA::InterfaceDef interface_def,
                    in boolean include_nested
                ) raises (
                    SysAdminException::ExNotFound
                );

        }; // End of PolicyDrivenBase interface

        interface PolicyRegion :
                PolicyDrivenBase,
                ManagedSets::FilteredSet,
                CosLifeCycle::GenericFactory
        {

                void add_instance_manager (
                    in ManagedInstances::InstanceManager im,
                    in Policies::InitializationPolicy initialization_policy,
                    in Policies::ValidationPolicy validation_policy
                ) raises (
                    SysAdminException::ExExists
                );

                void remove_instance_manager (
                    in ManagedInstances::InstanceManager im
                ) raises (
                    SysAdminException::ExObjNotFound,
                    SysAdminException::ExExists
                );

                void get_instance_manager_list (
                    in SelectionCriteria select
                    in unsigned long how_many;
                    out SysAdminTypes::ObjectLabelList ol_list;
                    out ManagedSets::ObjectLabelIterator iterator;
                );

                void set_initialization_policy (
                    in ManagedInstances::InstanceManager im,
                    in Policies::InitializationPolicy initialization_policy
                ) raises (
                    SysAdminException::ExObjNotFound
                );
```

```
SysAdminTypes::ObjectLabel get_initialization_policy (
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExObjNotFound
);

void set_validation_policy (
        in ManagedInstances::InstanceManager im,
        in Policies::ValidationPolicy validation_policy
) raises (
        SysAdminException::ExObjNotFound
);

SysAdminTypes::ObjectLabel get_validation_policy (
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExObjNotFound
);

void policy_validation (
        in ManagedInstances::InstanceManager im,
        in boolean enable
) raises (
        SysAdminException::ExObjNotFound
);

boolean is_validation_enabled (
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExObjNotFound
);

void verify_policy (
        in ManagedSets::Set scope,
        in boolean included_nested,
        in unsigned long how_many,
        out PolicyResultList pr_list,
        out PolicyResultIterator iterator
);

PolicyResultList get_policy_failures (
        in ManagedSets::Set scope,
        in boolean include_nested
        in unsigned long how_many,
        out PolicyResultList pr_list,
        out PolicyResultIterator iterator
);

SysAdminTypes::ObjectLabelList get_all_initialization_policies
(
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExNotFound
);

SysAdminTypes::ObjectLabelList get_all_enabled_validation_policies (
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExNotFound
```

```
            );

        }; // End of PolicyRegion interface

    }; // End of PolicyRegions module

    #endif // POLICYREGIONS_IDL
```

3.7.3 PolicyRegions::PolicyResultIterator

The **PolicyResultIterator** interface defines the following operations that allow an object to iteratively request a list of **PolicyResult** structures that were generated by verifying policy upon members of the current **PolicyRegion**. The **PolicyResultIterator** provides a static view of the policy conformance of the target members of the **PolicyRegion** at the time the iterator was created, and does not reflect subsequent changes to either the objects' membership in the **PolicyRegion** or their conformance to the current policies of the **PolicyRegion**. The operations provided are:

- *next_one*
- *next_n*
- *destroy*

3.7.3.1 The next_one Operation

The *next_one* operation returns the next **PolicyResult** structure. If the last **PolicyResult** structure is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_one (
        out PolicyResult pr;
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.3.2 The next_n Operation

The *next_n* operation returns at most the requested number of **PolicyResult** structures. If the last **PolicyResult** structure is being returned on this call (or was returned on a previous call), this operation returns FALSE, indicating subsequent calls to the iterator are not needed.

Syntax

```
boolean next_n(
        in unsigned long how_many,
        out PolicyResultList pr_list,
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.3.3 The destroy Operation

The *destroy* operation destroys the iterator.

Syntax

```
void destroy();
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.4 PolicyRegions::PolicyDrivenBase

The **PolicyDrivenBase** interface defines the operations required for an object to be managed by a policy region. It serves as a base interface to be inherited by interfaces for objects that will be managed and subject to policy.

The **PolicyRegions::PolicyDrivenBase** interface defines the following operations:

- *get_policy_region_info*
- *move_to_policy_region*
- *add_to_policy_region*
- *remove_from_policy_region*
- *list_enabled_validation_policies*
- *list_initialization_policies*

3.7.4.1 Inherited Interfaces

The **PolicyRegions::PolicyDrivenBase** interface inherits from the **ManagedInstances::Instance** interface.

3.7.4.2 The get_policy_region_info Operation

The *get_policy_region_info* operation returns information about the policy regions that the target object is a member of. This includes both the object reference and the label for each of the policy regions.

Syntax

```
SysAdminTypes::ObjectLabelList get_policy_region_info( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.4.3 The move_to_policy_region Operation

The *move_to_policy_region* operation moves the target **PolicyDrivenBase** object from the **PolicyRegion** object specified in the *pr_from* argument to the **PolicyRegion** object specified in the *pr_to* argument. The **InstanceManager** that the target object belongs to must be in the list of instance managers supported by the *pr_to* policy region. The target object must also be a member of the *pr_from* policy region.

Syntax

```
void move_to_policy_region(
        in PolicyRegions::PolicyRegion pr_from,
        in PolicyRegions::PolicyRegion pr_to
) raises (
        SysAdminException::ExObjNotFound,
        SysAdminException::ExNotFound,
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions, **SysAdminException::ExObjNotFound**, **SysAdminException::ExNotFound** and **SysAdminException::ExInvalid**.

SysAdminException::ExObjNotFound is raised when the target object is not currently a member of the *pr_from* policy region.

SysAdminException::ExNotFound is raised when the target object is being added to a policy region that does not support objects managed by the target object's instance manager.

SysAdminException::ExInvalid is raised when an implementation restriction on policy region membership would be violated by moving this **PolicyDrivenBase** object to this **PolicyRegion**.

SysAdminException::ExInvalid is raised when the label of the target object matches that of an object that is already in the **PolicyRegion** specified by *pr_to*, and the **CORBA::Object** defined *is_equivalent* method returns FALSE when invoked on the target object, passing the already existing **PolicyRegion** member. In this case the target object is still a member of the source **PolicyRegion**. If the **CORBA::Object** defined *is_equivalent* method returns TRUE in the situation described above, no exception is raised and there remains only one reference to the object in the **PolicyRegion**.

3.7.4.4 The add_to_policy_region Operation

The *add_to_policy_region* operation adds the target **PolicyDrivenBase** object to the **PolicyRegion** set specified in the *pr* argument. The **InstanceManager** that the target object belongs to must be in the list of instance managers supported by the policy region.

Syntax

```
void add_to_policy_region(
        in PolicyRegions::PolicyRegion pr
) raises (
        SysAdminException::ExNotFound,
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions, **SysAdminException::ExNotFound** and **SysAdminException::ExInvalid**.

SysAdminException::ExNotFound is raised when the target object is being added to a policy region that does not support objects managed by the target object's instance manager.

SysAdminException::ExInvalid is raised when an implementation restriction on policy region membership would be violated by adding this **PolicyDrivenBase** object to this **PolicyRegion**.

SysAdminException::ExInvalid is raised when the label of the target object matches that of an object that is already in the **PolicyRegion** specified by *pr*, and the **CORBA::Object** defined *is_equivalent* method returns FALSE when invoked on the target object, passing the already existing **PolicyRegion** member. If the CORBA::Object defined *is_equivalent* method returns TRUE in this case, no exception is raised and there remains only one reference to the object in the **PolicyRegion**.

3.7.4.5 The remove_from_policy_region Operation

The *remove_from_policy_region* operation removes the target **PolicyDrivenBase** object from the **PolicyRegion** set specified in the *pr* argument. If the specified policy region is the only policy region that the target object belongs to, the target object will not be removed.

Syntax

```
void remove_from_policy_region(
        in PolicyRegions::PolicyRegion pr
) raises (
        SysAdminException::ExObjNotFound,
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions, **SysAdminException::ExObjNotFound** and **SysAdminException::ExInvalid**.

SysAdminException::ExObjNotFound is raised when the target object is not currently a member of the specified policy region.

SysAdminException::ExInvalid is raised if the target object belongs to no other policy regions than the one specified in the *pr* argument.

3.7.4.6 The list_enabled_validation_policies Operation

The *list_enabled_validation_policies* operation returns a list of validation policies that the target policy driven base object is subject to. Only validation policies for which validation is enabled will be returned. The selection of the validation policies returned can be controlled through the arguments specified on the call to this operation.

The *policy_regions* argument is a list of policy regions of which the search is to be limited to. If the target policy driven base object is not a direct member of any of the listed policy regions, those policy regions are ignored in the search. If this argument is passed as an empty sequence (that is, sequence of zero length), all of the policy regions to which the target policy driven base object belongs will be included in the search.

The *include_nested* argument, if specified as FALSE, indicates that the search is to be limited to policy regions that the target object is directly a member of. If this argument is specified as TRUE, the search will include not only those policy regions to which the target policy driven base object is a direct member, but also all policy regions that are up the containment hierarchy from them.

The *interface_def* argument allows the validation policy objects being returned to be restricted to those that support the specified interface. If this argument is specified as NULL, all validation policy objects that meet the other selection criteria will be returned.

The validation policies returned will be ordered based on the following set of rules:

- If the *policy_regions* parameter is not specified, the order is undefined.
- If the *policy_regions* parameter is specified, the order will be the same as the ordering of policy regions within the parameter.
- If the *include_nested* parameter is specified as TRUE, then for each policy region in the *policy_regions* parameter, the topmost containing policy region's validation policy occurs prior to that of the policy region in the parameter[3].

Syntax

```
SysAdminTypes::ObjectLabelList list_enabled_validation_policies(
        in PolicyRegionList policy_regions,
        in CORBA::InterfaceDef interface_def,
        in boolean include_nested
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised when no validation policy objects meet the selection criteria specified on the call.

3.7.4.7 The list_initialization_policies Operation

The *list_initialization_policies* operation returns a list of initialization policies that the target policy driven base object is subject to. The selection of the initialization policies returned can be controlled through the arguments specified on the call to this operation.

The *policy_regions* argument is a list of policy regions of which the search is to be limited to. If the target policy driven base object is not a direct member of any of the listed policy regions, those policy regions are ignored in the search. If this argument is passed as an empty sequence (that is, sequence of zero length), all of the policy regions to which the target policy driven base object belongs will be included in the search.

The *include_nested* argument, if specified as FALSE, indicates that the search is to be limited to policy regions that the target object is directly a member of. If this argument is specified as TRUE, the search will include not only those policy regions to which the target policy driven base object is a direct member, but also all policy regions that are up the containment hierarchy from them.

The *interface_def* argument allows the initialization policy objects being returned to be restricted to those that support the specified interface. If this argument is specified as NULL, all initialization policy objects that meet the other selection criteria will be returned.

The initialization policies returned will be ordered based on the following set of rules:

3. It is possible for a policy region itself to be a member of multiple policy regions. When this occurs, the order of validation policies returned is not defined. Because of this, it is recommended that policy regions should not belong to more than one other policy region if deterministic ordering of validation policies is required.

- If the *policy_regions* parameter is not specified, the order is undefined.
- If the *policy_regions* parameter is specified, the order will be the same as the ordering of policy regions within the parameter.
- If the *include_nested* parameter is specified as TRUE, then for each policy region in the *policy_regions* parameter, the topmost containing policy region's initialization policy occurs prior to that of the policy region in the parameter[4].

Syntax

```
SysAdminTypes::ObjectLabelList list_initialization_policies(
        in PolicyRegionList policy_regions,
        in CORBA::InterfaceDef interface_def,
        in boolean include_nested
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised when no initialization policy objects meet the selection criteria specified on the call.

3.7.5 PolicyRegions::PolicyRegion Interface

The **PolicyRegions::PolicyRegion** interface defines the operations that provide the ability for system administration applications to group a set of managed resources and to define and uniformly apply the policies to the set of resources. In addition, operations that allow querying the policy region for the policy that applies to a particular instance manager are defined.

The **PolicyRegions::PolicyRegion** interface defines the following operations:

- *add_instance_manager*
- *remove_instance_manager*
- *get_instance_manager_list*
- *set_initialization_policy*
- *get_initialization_policy*
- *set_validation_policy*
- *get_validation_policy*
- *policy_validation*
- *is_validation_enabled*

4. It is possible for a policy region itself to be a member of multiple policy regions. When this occurs, the order of initialization policies returned is not defined. Because of this, it is recommended that policy regions should not belong to more than one other policy region if deterministic ordering of initialization policies is required.

- *verify_policy*
- *get_policy_failures*
- *get_all_initialization_policies*
- *get_all_enabled_validation_policies*

3.7.5.1 Inherited Interfaces

The **PolicyRegion** interface inherits from the following interfaces:

- **PolicyRegions::PolicyRegionBase**
- **CosLifeCycle::GenericFactory**
- **ManagedSets::FilteredSet**

Unique Behavior of Inherited Interfaces

Operations defined within these interfaces may have unique behavior when inherited by the **PolicyRegions::PolicyRegion** interface:

CosLifeCycle::LifeCycleObject

The implementation of the *remove* operation within the **PolicyRegions::PolicyRegion** interface must insure that the policy region is empty prior to deleting itself. If the policy region is not empty, an exception is raised.

CosLifeCycle::GenericFactory

PolicyRegion uses this interface to provide a convenience function, allowing other **PolicyRegion** objects similar to itself to be created and initialized.

The *k* argument is used with both the *create_object* and *supports* operations. It is defined to take the following:

kind **field**	*id* **field**
‘‘object interface’’	The fully scoped name of the principal IDL interface for the type of policy region to be created
‘‘object implementation’’	The name of the implementation to be used with the policy region to be created

For the *supports* operation the *k* argument is required. It is implementation dependent as to whether just ‘‘object interface’’ or ‘‘object implementation’’, or both, are required to be specified. If the **PolicyRegion** is capable of creating another policy region of the specified interface and implementation, it returns TRUE, otherwise it returns FALSE. Note that a **PolicyRegion** only creates objects of the same interface and implementation as itself.

For the *create_object* operation, the *k* argument is optional and may be specified as an empty sequence (that is, sequence of length zero). This is because the policy region already has a defined interface and implementation. If the *k* argument is specified, the values specified must be consistent with the interface and implementation of this policy region object. If it is not, the **CosLifeCycle::NoFactory** exception is raised.

The implementation of the *create_object* operation within the **PolicyRegions::PolicyRegion** interface creates another policy region that is contained in the creating region. That is, the new region is made a member of the policy region’s set. The new policy region is an exact duplicate of the owner, including the set of supported instance managers and policies, except that it is created empty of members. This is similar to calling the

ManagedInstances::PolicyRegionsInstanceManager specifying the target policy region object in the ‘‘policy regions’’ and ‘‘model policy region’’ name/value pairs of *the_criteria* argument.

For the *create_object* operation, after the new policy region is created and added to this policy region, the newly created policy region should be added to this policy region’s instance manager (which must support the **PolicyRegionsInstanceManager** interface).

The name/value pairs that are passed in *the_criteria* argument of the *create_object* call should contain the named values for *labelid* and *location*. These have the same meaning as defined for the **ManagedInstances::InstanceManager** interface. If these are not specified, the **CosLifeCycle::InvalidCriteria** exception is raised.

3.7.5.2 The add_instance_manager Operation

The *add_instance_manager* operation adds a new supported instance manager to the target object. The target object must be a policy region object. This operation optionally assigns an initialization policy object and validation policy object to the region. The *im* argument specifies the object reference of the instance manager to be added to the policy region. The *validation_policy* argument specifies the object reference of the validation policy to be assigned to the policy region. If this parameter is set to the object reference for a validation policy object, policy validation is enabled in the policy region for the instance manager. If this parameter is set to **OBJECT_NIL**, no policy validation is assigned to the new instance manager and policy validation is not enabled for it. The *initialization_policy* argument specifies the object reference of the initialization policy object to be assigned to the policy region. If this parameter is set to **OBJECT_NIL**, no initialization policy is assigned to the new instance manager and no initialization policy is applied. The policy objects specified by the *initialization_policy* and *validation_policy* parameters must already have been registered with the instance manager specified by the *im* parameter[5].

Syntax

```
void add_instance_manager(
        in ManagedInstances::InstanceManager im,
        in Policies::InitializationPolicy initialization_policy,
        in Policies::ValidationPolicy validation_policy
)raises(
        SysAdminException::ExExists
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExExists.**

SysAdminException::ExExists is raised when the specified instance manager is already a member of the policy region.

5. Refer to the operations of the **PolicyObjectAdmin** interface, described elsewhere in this specification.

3.7.5.3 The remove_instance_manager Operation

The *remove_instance_manager* operation removes the specified instance manager from the target object, which must support the **PolicyRegions::PolicyRegion** interface. There must be no members of the specified instance manager in the policy region for this operation to succeed. The *im* argument specifies the object reference of the instance manager object that manages the object type that is to be removed from the policy region.

Syntax

```
void remove_instance_manager(
        in ManagedInstances::InstanceManager im
)raises(
        SysAdminException::ExObjNotFound,
        SysAdminException::ExExists
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound** and **SysAdminException::ExExists**.

The **SysAdminException::ExObjNotFound** exception is raised if the instance manager object passed in the *im* parameter is not registered with this policy region.

The **SysAdminException::ExExists** exception is raised if this policy region currently has any policy driven base objects as members who are also members of the instance manager passed in the *im* parameter.

3.7.5.4 The get_instance_manager_list Operation

The *get_instance_manager_list* operation lists the instance manager supported by the target object, which must support the **PolicyRegions::PolicyRegion** interface. This operation lists the instance manager's object references and labels. The *select* argument specifies which of the instance managers supported by the target policy region will be returned in the list. The values for this argument have the following meaning:

all list all instance managers registered with the policy region

with_initialization
: list all instance managers that have an **InitializationPolicy** object registered with the policy region

with_validation
: list all instance managers that have a **ValidationPolicy** object registered with the policy region

with_validation_enabled
: list all instance managers that have validation enabled with the policy region

with_initialization_or_validation
: list all instance managers that have an **InitializationPolicy** object or **ValidationPolicy** object registered with the policy region

with_initialization_or_validation_enabled
: list all instance managers that have an **InitializationPolicy** object registered or have validation enabled with the policy region.

The initial size of *ol_list* is determined by the *how_many* argument. The rest of the **ObjectLabel** structures, if any, can be retrieved using the returned iterator object. When *how_many* is equal to or greater than the total number of **ObjectLabel** structures that could potentially be returned, all of the **ObjectLabel** structures will be returned in *ol_list* and the iterator will be returned as OBJECT_NIL.

Syntax

```
enum SelectionCriteria {
        all,
        with_initialization,
        with_validation,
        with_validation_enabled,
        with_initialization_or_validation,
        with_initialization_or_validation_enabled
};

void get_instance_manager_list (
        in SelectionCriteria select,
        in unsigned long how_many,
        out ObjectLabelList ol_list,
        out ObjectLabelIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.5.5 The set_initialization_policy Operation

The *set_initialization_policy* operation assigns the specified initialization policy object to the specified instance manager, replacing any previous assignment. This assignment is in effect only in the policy region represented by the target object. The initialization policy supplies the new instance manager members with initial attribute values. The policy region must already support the specified instance manager. Use the *add_instance_manager* operation to add a new supported instance manager to a policy region. The *im* argument is the object reference of the instance manager that defines the instance manager to which the initialization policy is to be assigned in the target policy region. The *initialization_policy* argument is the object reference of the initialization policy object whose methods return the proper default values. If this parameter is set to **OBJECT_NIL**, no initialization policy is assigned to the instance manager and the default initialization policy, if set, is applied.

Syntax

```
void set_initialization_policy(
        in Instances::InstanceManager im,
        in Policies::InitializationPolicy initialization_policy
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the instance manager to which the initialization policy is being set is not supported by the policy region.

3.7.5.6 The get_initialization_policy Operation

The *get_initialization_policy* operation returns the object reference of the initialization policy object assigned to the specified instance manager. The instance manager may have other initialization policy objects for other policy regions. The *im* argument is the object reference of the instance manager for the initialization policy object to be returned.

Syntax

```
SysAdminTypes::ObjectLabel get_initialization_policy (
        in ManagedInstances::InstanceManager im
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the specified instance manager does not have an initialization policy object registered for it.

3.7.5.7 The set_validation_policy Operation

The *set_validation_policy* operation assigns and enables the specified validation policy object to the specified instance manager, replacing any previous validation policy assignment. This assignment is in effect only in the policy region represented by the target object. The validation policy object supplies methods that validate changes to attributes defined by the instance manager. The *im* argument specifies the object reference of the instance manger to which the validation policy is to be assigned in the target policy region. If the *policy_validation* parameter is set to **OBJECT_NIL**, no validation policy is assigned to the instance manager and the default validation policy, if set, is used if enabled for it.

Syntax

```
void set_validation_policy (
        in ManagedInstances::InstanceManager im,
        in Policies::ValidationPolicy validation_policy
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the instance manager associated with the policy region.

3.7.5.8 The get_validation_policy Operation

The *get_validation_policy* returns the object reference of the validation policy object assigned to the specified instance manager. The instance manager may have other policy validation objects in other policy regions. The *im* argument is the object reference of the instance manager whose validation policy object reference is to be returned.

Syntax

```
SysAdminTypes::ObjectLabel get_validation_policy (
        in ManagedInstances::InstanceManager im
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when a validation policy object is not registered for the instance manager or the instance manager is not registered in the policy region.

3.7.5.9 The policy_validation Operation

The *policy_validation* operation enables or disables validation policy for the specified instance manager. This setting applies only to the specified instance manager in the target policy region. The instance manager can have validation policy independently enabled or disabled in other policy regions. The *im* argument is the object reference of the instance manager whose validation policy state is to be set. The enable argument is set to TRUE if policy validation is to be enabled for the specified instance manager. This argument is set to FALSE if validation policy is to be disabled.

Syntax

```
void policy_validation(
        in ManagedInstances::InstanceManager im,
        in boolean enable
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the specified instance manager is not supported by the policy region or if there is no **ValidationPolicy** object for the specified instance manager registered in the policy region.

3.7.5.10 The is_validation_enabled Operation

The *is_validation_enabled* operation returns TRUE or FALSE, depending on whether validation policy is enabled for the instance manager defined by the *im* argument. If there is no validation policy object for the instance manager, the method returns FALSE.

Syntax

```
boolean is_validation_enabled(
        in ManagedInstances::InstanceManager im
) raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the specified instance manager is not supported by the policy region.

3.7.5.11 The verify_policy Operation

The *verify_policy* operation is used to verify if objects within the target policy region adhere to the policy for the region. This operation provides positive feedback, with information being returned about every object that was checked, irrespective of whether it passed or failed validation. The returned **PolicyResultList** contains a **PolicyResult** structure for every object that was checked, and the *passed_policy* boolean in that structure can be used to see if the object passed or failed validation. It is the *validate_policy_driven_object* operation defined on the **Policies::ValidationPolicy** object that is used to perform the verification.

The operation may include all objects within the policy region or be scoped to only include objects within the region that are also members of a set passed in the *scope* argument. Note that if OBJECT_NIL is passed as the *scope* argument, or if the object reference of the target policy region itself is passed as the *scope* argument, all of the members of the policy region will be checked.

If the *include_nested* boolean is set to TRUE, then policy defined in all of the target policy region's containing policy regions will be used to validate the objects as well.

For any object whose instance manager does not currently have validation enabled within a policy region, no checking is done in respect to policy for that object relative to that region.

Since each object can be checked multiple times when *include_nested* is TRUE, it is possible for there to be multiple **PolicyResult** structures in the **PolicyResultList** for any one object in the target policy region. The **PolicyResult** structure for each validation that was performed contains:

- An object reference to the object that was checked
- An object reference to the policy region within which the policy is defined
- An object reference to the validation policy object that was used to perform the validation
- A boolean indicator as to whether or not the object passed validation
- A string with a description of the reason for validation failure

This provides sufficient information for the caller to determine exactly what was validated and from where any validation failures originated.

The initial size of *pr_list* is determined by the *how_many* argument. The rest of the **PolicyResult** structures, if any, can be retrieved using the returned *iterator* object. When *how_many* is equal to or greater than the total number of **PolicyResult** structures that could potentially be returned, all of the **PolicyResult** structures will be returned in *pr_list* and the *iterator* will be returned as OBJECT_NIL.

Syntax

```
PolicyResultList verify_policy(
        in ManagedSets::Set scope,
        in boolean include_nested
        in unsigned long how_many,
        out PolicyResultList pr_list,
        out PolicyResultIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.5.12 The get_policy_failures Operation

The *get_policy_failures* operation is used to verify if objects within the target policy region adhere to the policy for the region. This operation provides feedback only for objects that failed validation. The returned **PolicyResultList** contains a **PolicyResult** structure for every object that failed validation. It is the *validate_policy_driven_object* operation defined on the **Policies::ValidationPolicy** object that is used to perform the verification.

The operation may include all objects within the policy region or be scoped to only include objects within the region that are also members of a set passed in the *scope* argument. Note that if OBJECT_NIL is passed as the *scope* argument, or if the object reference of the target policy region itself is passed as the *scope* argument, all of the members of the policy region will be checked.

If the *include_nested* boolean is set to TRUE, then policy defined in all of the target policy region's containing policy regions will be used to validate the objects as well.

For any object whose instance manager does not currently have validation enabled within a policy region, no checking is done in respect to policy for that object relative to that region.

Since each object can be checked multiple times when *include_nested* is TRUE, it is possible for there to be multiple **PolicyResult** structures in the **PolicyResultList** for any one object in the target policy region, assuming it failed validation in respect to more than one region. The **PolicyResult** structure for each failure contains:

- An object reference to the object that was checked
- An object reference to the policy region within which the policy is defined
- An object reference to the validation policy object that was used to perform the validation
- A boolean indicator set to FALSE
- A string with a description of the reason for validation failure

This provides sufficient information for the caller to determine exactly where any validation failures originated from.

The initial size of *pr_list* is determined by the *how_many* argument. The rest of the **PolicyResult** structures, if any, can be retrieved using the returned *iterator* object. When *how_many* is equal to or greater than the total number of **PolicyResult** structures that could potentially be returned, all of the **PolicyResult** structures will be returned in *pr_list* and the *iterator* will be returned as OBJECT_NIL.

Syntax

```
PolicyResultList get_policy_failures(
        in ManagedSets::Set scope,
        in boolean include_nested
        in unsigned long how_many,
        out PolicyResultList pr_list,
        out PolicyResultIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.7.5.13 The get_all_initialization_policies Operation

The *get_all_initialization_policies* operation returns a list of initialization policy objects, within the target policy region and all of it's containing policy regions, that are assigned to the instance manager specified in the *im* argument. The initialization policies returned will be ordered in the same order as the nesting of policy regions, with the topmost containing policy region's initialization policy occurring first in the sequence and the target policy region's initialization policy occurring last in the sequence[6].

Syntax

```
SysAdminTypes::ObjectLabelList get_all_initialization_policies(
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised when no initialization policy objects are found that satisfy the request.

3.7.5.14 The get_all_enabled_validation_policies Operation

The *get_all_enabled_validation_policies* operation returns a list of validation policy objects, within the target policy region and all of it's containing policy regions, that are assigned to the instance manager specified in the *im* argument. Only those validation policy objects for which validation is enabled are returned. The validation policies returned will be ordered in the same order as the nesting of policy regions, with the topmost containing policy region's validation policy occurring first in the sequence and the target policy region's validation policy occurring last in the sequence[7].

---

6. It is possible for a policy region itself to be a member of multiple policy regions. When this occurs, the order of initialization policies returned is not defined. Because of this, it is recommended that policy regions should not belong to more than one other policy region if deterministic ordering of initialization policies is required."

7. It is possible for a policy region itself to be a member of multiple policy regions. When this occurs, the order of validation policies returned is not defined. Because of this, it is recommended that policy regions should not belong to more than one other policy region if deterministic ordering of validation policies is required.

Syntax

```
SysAdminTypes::ObjectLabelList get_all_enabled_validation_policies(
        in ManagedInstances::InstanceManager im
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised when no validation policy objects are found that satisfy the request.

3.8 Policies Module

The **Policies** module defines the interfaces related to policy objects. These interfaces support interaction between policy objects and objects supporting the **PolicyRegion** or the **InstanceManager** interface.

3.8.1 Interfaces and Operations

Module	Interface	Operation
Policies	PolicyObject *Inheritance:* Identification::Labeled	*get_policy_driven_object_type*
	PolicyObjectAdmin	*get_initialization_policies* *get_default_initialization* *get_validation_policies* *get_default_validation* *add_initialization* *set_default_initialization* *remove_initialization* *add_validation* *remove_validation* *set_default_validation* *add_pr_backref* *remove_pr_backref* *get_pr_backrefs*
	InitializationPolicy *Inheritance:* Policies::PolicyObject	*initialize_policy_driven_object*
	ValidationPolicy *Inheritance:* Policies::PolicyObject	*validate_policy_driven_object*

Table 3-6 Interfaces and Operations for the Policies Module

3.8.2 Specified IDL

```
//
// Component Name: Policies.idl
//
// Description:
// The following interfaces provide the functionality
// for support of policy objects.
//

#ifndef POLICIES_IDL
#define POLICIES_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <Identification.idl>
```

```
#include <CosLifeCycle.idl>
#include <ManagedSets.idl>

module Policies {

        // forward references
        interface PolicyObject;
        interface PolicyObjectAdmin;
        interface InitializationPolicy;
        interface ValidationPolicy;

        interface PolicyObjectAdmin
        {

                SysAdminTypes::ObjectLabelList get_initialization_policies();

                InitializationPolicy get_default_initialization (
                ) raises (
                        SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList get_validation_policies();

                ValidationPolicy get_default_validation (
                ) raises (
                        SysAdminException::ExNotFound
                );

                void add_initialization (
                        in InitializationPolicy initialization_policy
                ) raises (
                        SysAdminException::ExExists,
                        SysAdminException::ExInvalid
                );

                void set_default_initialization (
                        in InitializationPolicy initialization_policy
                ) raises (
                        SysAdminException::ExInvalid
                );

                void remove_initialization (
                        in InitializationPolicy initialization_policy
                ) raises (
                        SysAdminException::ExObjNotFound
                );

                void add_validation (
                        in ValidationPolicy validation_policy
                ) raises (
                        SysAdminException::ExExists,
                        SysAdminException::ExInvalid
                );

                void remove_validation (
                        in ValidationPolicy validation_policy
                ) raises (
                        SysAdminException::ExObjNotFound
                );
```

```
            void set_default_validation (
                    in ValidationPolicy validation_policy
            ) raises (
                    SysAdminException::ExInvalid
            );

            void add_pr_backref (
                    in PolicyRegion pr
            );

            void remove_pr_backref (
                    in PolicyRegion pr
            ) raises (
                    SysAdminException::ExNotFound
            );

            void get_pr_backrefs (
                    in unsigned long how_many,
                    out SetList s_list,
                    out SetIterator iterator
            );

};  // End of PolicyObjectAdmin interface

// The PolicyObject interface defines the common
// requirements of policy objects

interface PolicyObject :
            Identification::Labeled
{
            string get_policy_driven_object_type();

            CORBA::InterfaceDef get_policy_driven_object_interface();

};  // End of PolicyObject interface

// The InitializationPolicy interface defines the
// requirements of an initialization policy object

interface InitializationPolicy :
            PolicyObject
{
            void initialize_policy_driven_object (
                    in Object object_to_initialize,
                    in CosLifeCycle::Criteria override_criteria
            );

};  // End of InitializationPolicy interface

// The ValidationPolicy interface defines the
// requirements of an validation policy object

interface ValidationPolicy :
            PolicyObject
{

            boolean validate_policy_driven_object (
                    in Object object_to_validate,
                    out string description
```

```
        );

    }; // End of ValidationPolicy interface

}; // End of Policies module

#endif // POLICIES_IDL
```

3.8.3 Policies::PolicyObjectAdmin Interface

The **Policies::PolicyObjectAdmin** interface defines operations that all instance managers that have policy associated with them must support. These operations allow the reporting of the policy objects associated with the establishment and enforcement of policy, along with the administration of these policy objects.

The **Policies::PolicyObjectAdmin** interface defines the following operations:

- *get_initialization_policies*
- *get_default_initialization*
- *get_validation_policies*
- *get_default_validation*
- *add_initialization*
- *set_default_initialization*
- *remove_initialization*
- *add_validation*
- *remove_validation*
- *set_default_validation*

3.8.3.1 The get_initialization_policies Operation

The *get_initialization_policies* operation returns the object references and labels of the initialization policy objects associated with the instance manager.

Syntax

```
SysAdminTypes::ObjectLabelList get_initialization_policies( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.3.2 The get_default_initialization Operation

The *get_default_initialization* operation returns the default initialization policy object for the instance manager. If no initialization policy has been explicitly set for the target instance manager in the current policy region, then the default initialization policy will be used to initialize new policy-driven objects during object creation.

Syntax

```
InitializationPolicy get_default_initialization(
)raises(
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised if a initialization policy object is not found for the object.

3.8.3.3 The get_validation_policies Operation

The *get_validation_policies* operation returns the object references and labels of any validation policy objects associated with the instance manager.

Syntax

```
SysAdminTypes::ObjectLabelList get_validation_policies( );
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.3.4 The get_default_validation Operation

The *get_default_validation* operation returns the default validation policy object for the instance manager. If no validation policy has been explicitly set for the target instance manager in the current policy region, then the default validation policy will be used to validate policy-driven objects if policy validation is enabled for the instance manager in the policy region of interest.

Syntax

```
ValidationPolicy get_default_validation(
)raises(
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised if a validation policy object is not registered for the object.

3.8.3.5 The add_initialization Operation

The *add_initialization* operation adds a reference to a new initialization policy object for the instance manager. *Object::_nil()* is not permitted as an input value.

Syntax

```
void add_initialization(
        in InitializationPolicy initialization_policy
)raises(
        SysAdminException::ExExists,
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExExists**.

SysAdminException::ExExists is raised if the specified object is already registered for the instance manager.

SysAdminException::ExInvalid is raised if the specified object indicates that it supports a policy driven object type that is not an interface supported by objects managed by the receiving instance manager.

3.8.3.6 The set_default_initialization Operation

The *set_default_initialization* operation sets the default initialization policy object for the instance manager. If the input initialization policy object is not associated with the instance manager, then the initialization policy object will be added to the instance manager automatically before *set_initialization_policy* returns control to the caller. If the value of the input *initialization_policy* is *Object::_nil()*, then the default initialization policy is set to *Object::_nil()* and there is effectively no default initialization policy for the instance manager.

Syntax

```
void set_default_initialization(
        in InitializationPolicy initialization_policy
)raises(
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExInvalid**.

SysAdminException::ExInvalid is raised if the input initialization policy object indicates that it supports a policy driven object type that is not an interface supported by objects managed by the receiving instance manager.

3.8.3.7 The remove_initialization Operation

The *remove_initialization* operation removes the reference to the specified initialization policy object from the instance manager. *Object::_nil()* is not permitted as an input value.

If the input initialization policy happens to be the instance manager's default initialization policy, then *remove_initialization* should set the instance manager's default initialization policy to *Object::_nil()*.

Syntax

```
void remove_initialization(
        in InitializationPolicy initialization_policy
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the requested initialization policy object is not currently associated with the instance manager.

3.8.3.8 The add_validation Operation

The *add_validation* operation adds a reference to a new validation policy object for the instance manager. *Object::_nil()* is not permitted as an input value.

Syntax

```
void add_validation(
        in ValidationPolicy validation_policy
)raises(
        SysAdminException::ExExists,
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExExists**.

SysAdminException::ExExists is raised if the specified validation policy object is already associated with the instance manager.

SysAdminException::ExInvalid is raised if the specified object indicates that it supports a policy driven object type that is not an interface supported by objects managed by the receiving instance manager.

3.8.3.9 The remove_validation Operation

The *remove_validation* operation removes a reference to a validation policy object from the instance manager. *Object::_nil()* is not permitted as an input value.

If the input validation policy happens to be the instance manager's default validation policy, then *remove_validation* should set the instance manager's default validation policy to *Object::_nil()*.

Syntax

```
void remove_validation(
        in ValidationPolicy validation_policy
)raises(
        SysAdminException::ExObjNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExObjNotFound**.

SysAdminException::ExObjNotFound is raised when the requested validation policy object is not currently associated with the instance manager.

3.8.3.10 The set_default_validation Operation

The *set_default_validation* operation sets the default validation policy object for the instance manager. If the input validation policy object parameter is not associated with the instance manager, then the validation policy object will be added to the instance manager automatically before *set_default_validation* returns control to the caller. If the value of the input *initialization_policy* is *Object::_nil()*, then the default initialization policy is set to *Object::_nil()* and there is effectively no default initialization policy for the instance manager.

Syntax

```
void set_default_validation(
        in ValidationPolicy validation_policy
)raises(
        SysAdminException::ExInvalid
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExInvalid**.

SysAdminException::ExInvalid is raised if the input validation policy object indicates that it supports a policy driven object type that is not an interface supported by objects managed by the receiving instance manager.

3.8.3.11 The add_pr_backref Operation

The *add_pr_backref* operation adds a reference from the target object to a policy region. The purpose of this operation is to allow instance managers to maintain references to the policy regions that support them. The intended usage is that this operation should only be invoked within the implementation of the policy region's *add_instance_manager* operation, and not by an arbitrary client.

Syntax

```
void add_pr_backref (
        in PolicyRegion pr
);
```

Exceptions

CORBA 1.2 standard exceptions

3.8.3.12 The remove_pr_backref Operation

The *remove_pr_backref* operation removes a reference from the target object to a policy region. The intended usage is that this operation should only be invoked within the implementation of the policy region's *remove_instance_manager* operation, and not by an arbitrary client.

Syntax

```
void remove_pr_backref (
        in PolicyRegion pr
) raises (
        SysAdminException::ExNotFound
);
```

Exceptions

CORBA 1.2 standard exceptions and **SysAdminException::ExNotFound**.

SysAdminException::ExNotFound is raised if the target instance manager does not currently have a back reference to the policy region passed in as an input parameter.

3.8.3.13 The get_pr_backrefs Operation

The *get_pr_backrefs* operation returns as an output parameter a list of references to the policy regions by which the target object is currently supported. It accepts an input parameter, *how_many*, that designates the maximum desired number of such references to return in the list. The list of returned references is contained in the first output parameter. If the number of policy regions that the target object is currently supported by is great than *how_many*, a Set iterator object is created as a bi-product of this operation and contained in the second output parameter. This iterator can be used to iterate through the additional supporting policy regions not returned in the first output parameter, using *SetIterator* operations as previously described in this specification.

Syntax

```
void get_pr_backrefs (
        in unsigned long how_many,
        out SetList s_list,
        out SetIterator iterator
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.4 Policies::PolicyObject Interface

The **Policies::PolicyObject** interface defines operations that all policy objects support. It serves as a base interface definition from which all other policy interfaces are derived.

The **Policies::PolicyObject** interface defines a single operation, *get_policy_driven_object_type*.

3.8.4.1 Inherited Interfaces

The **Policies::PolicyObject** interface inherits from the **Identification::Labeled** interface to ensure that all policy objects have a label.

3.8.4.2 The get_policy_driven_object_type Operation

The *get_policy_driven_object_type* operation returns the type name that describes an interface supported by the policy driven objects that this policy object can be associated with. Note that the type does not have to be the principal interface of the policy driven object, only a supported interface.

Syntax

```
string get_policy_driven_object_type();
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.4.3 The get_policy_driven_object_interface() Operation

The **get_policy_driven_object_interface** ( ) operation returns the **CORBA::InterfaceDef** that describes an interface supported by the policy-driven objects with which this policy object can be associated. Note that this type does not have to be the principal interface of the policy-driven object, only a supported interface.

Syntax

```
CORBA::InterfaceDef get_policy_driven_object_interface();
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.5 Policies::InitializationPolicy Interface

The **Policies::InitializationPolicy** interface defines a single operation, *initialize_policy_driven_object* .

3.8.5.1 The initialize_policy_driven_object Operation

The *initialize_policy_driven_object* initializes a single policy-driven object. This operation is intended to be called during the processing of the *create_object* operation by an instance manager, rather than being called directly by a client. In addition to accepting the object to be initialized as an input, *initialize_policy_driven_object* also accepts a sequence of name/value pairs or *criteria.* The criteria can be used to override the default values that will be used to initialize the object at creation time. This allows instance specific initialization data to be combined with the policy region specific data (as embodied in the InitializationPolicy object) to define the initial object values.

It is expected that for every specialization of the *PolicyDrivenBase* type, there will be an implementation of this operation that specifically knows how to initialize objects of that type. Also, the definition of what name/value pairs are passed in the *override_criteria* parameter will typically be unique to each implementation.

Syntax

```
void initialize_policy_driven_object(
        in Object object_to_initialize,
        in CosLifeCycle::Criteria override_criteria
);
```

Exceptions

CORBA 1.2 standard exceptions.

3.8.6 Policies::ValidationPolicy Interface

The **Policies::ValidationPolicy** interface defines a single operation, *validate_policy_driven_object* .

3.8.6.1 The validate_policy_driven_object Operation

The *validate_policy_driven_object* validates a single policy-driven object. This operation is intended to be called by the *verify_policy* or *get_policy_failures* operations on a policy region or it can also be called directly by client code. It is expected that for every specialization of the *PolicyDrivenBase* type, there will be an implementation of this operation that specifically knows how to validate objects of that type. If the object conforms to the policy embodied by this *ValidationPolicy* object, the operation returns TRUE, otherwise it returns FALSE. When the return value is FALSE, the *description* out parameter contains a string indicating the reason the object did not conform to policy.

Syntax

```
boolean validate_policy_driven_object(
        in Object object_to_validate,
        out string description
);
```

Exceptions

CORBA 1.2 standard exceptions.

Chapter 4

Command Line Interface

In order to provide the developer and the end-user system administrator with the ability to write implementations of methods and client programs in shell or other interpreted languages, several general purpose commands are defined. This effectively specifies a shell binding to OMG IDL.

One goal of a means to invoke operations defined in IDL using a shell script is to provide access to as many IDL operations as possible. Initially client programs are written for each IDL operation desired. This has the unfortunate side effect of requiring many executables in an environment, and requiring system administration environment providers to anticipate what an end user of the environment would like to do from the command line. Another goal of the effort is to provide a general purpose mapping that is easily parsed by a shell script and was natural for a shell programmer. The shell model of interacting with an ORB is to present ORB methods as programs that communicate using standard input and standard output. A consequence of this idiom is that any shell script that has side effects on standard input or standard output will break the linkage between the callee and the caller. Thus it is part of the IDL contract that standard input and standard output are not affected by side effects arising from the method being called.

The shell programmer should deal naturally with the data being used. For example, an instance of an IDL struct:

```
struct X {        long a;   string b;        short c; };
```

might look like the following to a shell programmer:

```
{ 100000 "now is the time to do something" 20 }
```

A sequence of struct X might be:

```
{2 {100000 "now is the time ..." 20}
  {900000 "now it is too late" 40}}
```

To do this, a program is needed that can turn a client's input into this format. Conversely, a program is also needed that can take this as input and generate the appropriate ''idl'' invocation. The program will use the operation's type signature (possibly from an interface repository) to get rules for parsing the input and output.

If the needs of a shell script writer are examined, one can see how a script might be organized to provide the necessary functionality. First, the shell script writer needs access to the input arguments in some kind of native format. In the previous example, if a struct X is the input argument, the writer needs the struct in a string-like format that can be used and manipulated. Shells work with strings and shell variables - this is what is needed. If there are multiple arguments, the shell script writer needs to be able to get to each argument individually or as a group - much like **$0** or **$n** in **awk**. Secondly, the shell script writer needs to pass values to other methods and receive their results. Thirdly, the shell script writers needs to return output arguments in a similar fashion to receiving input arguments. To do this requires some order on how results are returned.

Presume the signatures:

```
interface Intf1 {
        long foo(
                in Object tgt,
                in X x,
                inout string y,
                out X z
        );
};
interface Intf2 {
        long bar(
                inout X x,
                out string y
        );
};
```

The following script shows how operation foo is implemented using a shell script, and how the shell implementation can use bar.

```
#!/bin/sh
#
# interface Intf1 {
#       long foo (
#               in Object tgt,
#               in X x,
#               inout string y,
#               out X z
#       );
# }

# Get all of the idl input arguments into a shell variable
my_sigs = ‘idllookup Intf1‘

# Alternatively, the signature could have been defined in an
# included file generated by an idl compiler.

inargs=‘idlinput $my_sigs‘

oper=‘idlarg 1 $inargs‘
self=‘idlarg 2 $inargs‘
if [ $oper -ne "Intf1::foo" ]; then
        idlexception {{tk_struct BAD_OPERATION}}
        exit 1
fi

tgt=‘idlarg 3 $inargs‘
x=‘idlarg 4 $inargs‘
y=‘idlarg 5 $inargs‘

# Call the bar method
# The target signature could be defined (perhaps in a file
# generated by an idl compiler) as
#
#  tgt_sig = "tk_long Intf2::bar {
#               inout   x $X_tc
#               out     x tk_string
#       }"
#
```

```
# or it could be retrieved from the interface repository via
#
#       tgt_sig='idllookup Intf2::bar'
#
# See section 4.3.1 for the format of signatures.

bar_results='idlcall $tgt $tgt_sig <<!EOF
$x
!EOF
`
if [ $? -ne 0 ]; then
# if an exception, the results have the stringified exception data
        idlexception $bar_results
        exit 1
fi

# now fetch the result arguments from bar's result string.
bar_long=`idlarg 3 $bar_results`
x=`idlarg 1 $bar_results`
bar_str=`idlarg 2 $bar_results`

# Pass the bar output back as the result to the caller.  Note
# that the order of the results is important (first the
# out/inout arguments in the same order as the IDL signature,
# then the result (if any).
out_args='idlresult ${mysig[0]} << !EOF
$bar_str
$x
$bar_long
!EOF
`
echo $out_args

exit 0
```

In the above script, the typecodes are directly supplied for the **idlinput**, **idlcall** and **idlresult** calls. If a signature passed into a command has the form:

```
{tk_indirect op-name}
```

then the command attempts to find the appropriate signature (perhaps using the interface repository).

4.1 Type Mappings

All types in the string-based binding must be represented by strings. The same rules that apply to manipulating strings in the Bourne shell apply here. The strings are delimited by spaces, not commas.

All scalar types will be represented by a string. Boolean *true* is represented by TRUE and *false* is represented by FALSE. All constructed types (array, sequence, struct, union) are described by listing their members enclosed within a pair of { }. The following are more explicit rules.

- **array**
  List all of the elements starting at the first index.
- **sequence**
  List the length, followed by the members.

The number of members listed must match the length. If the length is zero, no members are listed.

- **struct**
  List the members in the order of its IDL description.
- **union**
  List the discriminator followed by the value.
- **object**
  List the stringified representation (Object_to_string).
- **TypeCode**
  List the ‘‘kind’’ (tk_long, tk_objref, or tk_sequence, ...) followed by the parameters for the typecode. Parameters for a typecode can be found in CORBA 1.2 Table 16 (reproduced here):

  tk_null
  tk_void
  tk_short
  tk_long
  tk_ushort
  tk_ulong
  tk_float
  tk_double
  tk_boolean
  tk_char
  tk_octet
  tk_string
  tk_any
  tk_TypeCode
  tk_Principal
  {tk_objref interface-id}
  {tk_struct struct-name member-name TypeCode, ... (repeat pairs)}
  {tk_exception exception-name member-name TypeCode, ... (repeat pairs)}
  {tk_union union-name switch-TypeCode label-value member-name TypeCode, ... (repeat triples)}
  {tk_enum enum-name enum-id ...}
  {tk_sequence typecode maxlen-integer}
  {tk_array typecode length-integer}

  Note: **interface-id** , **struct-name** , **exception-name** , **union-name** and **enum-name** are fully scoped. **member-names** and **enum-ids** are not.
- **enum**
  List the enum identifier.
- **any**
  List the TypeCode followed by the value. For example:

  {tk_long 10}
  {{tk_struct X a tk_long b tk_string} {10 "hello world"}}
- **exception**
  The data is returned as the exception type, the exception identifier, and the exception value as a struct (defined previously).

4.2 Argument Ordering

When an IDL interface is *mapped* to this binding, on invocation only the *in* and *inout* arguments are placed in the input stream to the command. Thus, all *out* parameters are omitted during the invocation. When results are returned, the *out* and *inout* parameters and then return value are placed in the results stream. The *out* and *inout* parameters are in the same order of appearance as the IDL definition of the operation. As an example, if the IDL definition of an operation is as follows:

```
long op1(in long a1, out long a2, inout long a3, out long a4);
```

and **$A1** is the string version of **a1**, and **$A2** the string version of **a2**, etc., then to call **op1** from a shell the command would be:

```
# target signature obtained directly (or via compiler generated code):
#
#       tgt_sig = "tk_long Intf3::op1 {
#                               in    a1 tk_long
#                               out   a2 tk_long
#                               inout a3 tk_long
#                               out   a4 tk_long
#                               }"
#
# or via the interface repository:
#
#       tgt_sig = `idllookup Intf3::op1`

$op1_results=idlcall $tgt $tgt_sig << !EOF
$A1
$A3
!EOF
T
```

The results stream, contained in **op1_results**, will have the string version of **a2**, **a3** and **a4**, then the long specified by the return value of operation **op1**.

4.3 Defined Commands

This section defines the commands necessary for the invocation of operations, receipt of input, output and exceptions in natural shell style, and the parsing of arguments in a shell variable.

4.3.1 The idlcall Command

The **idlcall** command has the following syntax:

```
idlcall target signature [arg1 [arg2 ...]]
```

This command may accept arguments either from the command line or from standard input. If any arguments (that is, those items marked arg[n] in the above syntax) are specified on the command line, then standard input will be ignored. This command will invoke the IDL defined operation described by **signature** on the object represented by target: **target** is a *stringified* object reference. An IDL attribute X is mapped to the methods ‘‘_get_X’’ and ‘‘_set_X’’. The initial ‘‘_’’ ensures that these methods will not collide with method names that may be explicitly declared in the interface. Read-only attributes do not have a ‘‘_set_X’’ method. The arguments passed will be converted into the proper in-memory format, as defined in the IDL interface, in their order in the argument list. Only *in* and **inout** parameters should be present in the parameter list. The results will be returned with the return parameter of the operation as the first element in the

results and the remainder in the order of their position in the parameter list. Only **inout** and **out** parameters are returned in addition to results. The resulting strings are written to standard output.

If the invocation of the operation results in the raising of an exception, only exception information will be contained in the results passed to standard out. When an exception is raised, the **idlcall** command exits non-zero. An exception is returned in the result parameter as an Any (that is, {type-code value}).

The *signature* used in the call has the following form:

```
{return-typecode operation-name
{direction param-name parm-typecode ... (repeat triple)}
{exception-typecode ... (list all exceptions)}}
```

Signatures may be specified explicitly or may be obtained via **idllookup** (see Section 4.3.6 on page 99. It is expected that explicitly specified signatures will be obtained from included files emitted by idl compilers.

If a signature has the form:

```
{"tk_indirect" operation-name},
```

then the actual signature will be supplied by the command (possibly using an interface repository).

4.3.2 The idlinput Command

The **idlinput** command has the following syntax:

```
idlinput signature-list
```

This command returns the input passed to an operation implemented in shell and converts it to string form for processing/parsing in the shell implementation. The results of the conversion are written to standard output.

The *signature-list* is a list of method signatures (including ‘‘_get_X ’’and ‘‘_set_X’’ methods associated with attributes) that the shell script implements. The result of this call will be of the form:

```
{operation-name object-id in-arg1 in-arg2 ...}
```

arg 1 of the results that appear on stdout will be the *stringified* object reference for the current shell; *arg 2* of the results will be the the method actually being invoked.

4.3.3 The idlarg Command

The **idlarg** command has the following syntax:

```
idlarg op_name parameter_num arg
```

This command accepts the operation name, **op_name**, the numeric 1-based position of the desired attribute and the parameters. The position is the number of the parameter in the argument list, that is, the first parameter is in position 1, the second in position 2, etc., **parameters** contains the formatted string of parameters. Please note that the **operation_name** and the **object_id** are returned as the first 2 parameters from the *idl_input* command.

This command is also capable of extracting particular items or elements of structures, sequences, etc. This is achieved by extracting the complex type using **idlarg** and then using **idlarg** to extract components of the complex type.

4.3.4 The idlresults Command

The **idlresults** command has the following syntax:

```
idlresults signature [results]
```

This command returns the results of the method specified by the *signature*, whose format is defined in Section 4.3.1 on page 97, to a shell implementation. The format of the results is the string representation of the return value and the **inout**/**out** parameters of the IDL operation. The results strings are read from standard input or are provided in an argument list on the command line.

4.3.5 The idlexception Command

The **idlexception** command has the following syntax:

```
idlexception results-as-any
```

This command extracts the string form of the exception data from **results**, and writes this information to standard output.

4.3.6 The idllookup Command

The idllookup command has the following syntax:

```
idllookup -s name

idllookup -t name
```

The first form returns the signature of **name**, which may be a fully-qualified interface name, operation name, or attribute name. For an interface name, the result of the call is a signature list for the interface, suitable for passing to **idlinput**. For an operation name, the result of the call is the signature of the operation, suitable for passing to **idlcall** or **idlresult**. For an operation name, the result is the two operation signatures describing the attribute's ''get'' and ''set'' methods. A read-only attribute will return a single signature, for the ''get'' method.

The second form returns the typecode associated with **name**, which may be a fully-qualified type (including interface) name or interface name.

The two forms are needed to resolve the ambiguity in the case of interfaces and attributes.

The operation is undefined when called with the fully-qualified name of a module.

Appendix A

IDL Definitions for Management Facilities Interfaces

This Appendix provides interface definitions that can be compiled for the system management interfaces defined within this specification. If there is a conflict between the IDL found in this appendix and the IDL found in Chapter 3, the IDL found in Chapter 3 should be assumed to be correct.

A.1 SysAdminTypes.idl

```
//
// Component Name: SysAdminTypes.idl
//
// Description:
// The SysAdminTypes file defines types and data
// structures that are used frequently throughout the
// development of system administration applications.
//

#ifndef SYSADMINTYPES_IDL
#define SYSADMINTYPES_IDL

#include <orb.idl>
#include <CosNaming.idl>

module SysAdminTypes {

        typedef sequence <CORBA::InterfaceDef> InterfaceDefList;

        typedef CosNaming::NameComponent LabelType;

        struct ObjectLabel {
                Object objref;
                LabelType label;
        };

        struct LabelExpression {
                string id_regex;
                string kind_regex;
        };

        typedef sequence <ObjectLabel> ObjectLabelList;

        // The Platform structure defines elements for information
        // related to the hardware and operating system of a
        // client machine. The elements represent the following
        // information.

        struct Platform {
                string host_name;
                string machine_hardware_name;
                string operating_system_name;
                string operating_system_version;
                string operating_system_release;
        };

};

#endif //SYSADMINTYPES_IDL
```

A.2 Identification.idl

```
//
// Component Name: Identification.idl
//
// Description:
// This module defines the methods that implement an object's
// label, which is a name that uniquely identifies the object
// within an environment.
//

#ifndef IDENTIFICATION_IDL
#define IDENTIFICATION_IDL

#include <SysAdminTypes.idl>

module Identification {

        interface Labeled {

                SysAdminTypes::LabelType get_label();

                void set_label (in SysAdminTypes::LabelType label);

        };
};

#endif //IDENTIFICATION_IDL
```

A.3 ManagedSets.idl

```
//
// Component Name: ManagedSets.idl
//
// Description:
// The following interfaces provide the functionality
// for support of managed sets.
//

#ifndef MANAGEDSETS_IDL
#define MANAGEDSETS_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <Identification.idl>

module ManagedSets {

//-------------------------------------------------------------------
//Forward references
//-------------------------------------------------------------------
interface Member;
interface SetIterator;
interface MemberIterator;
interface MemberLabelIterator;
interface Set;
interface FilteredSet;

//-------------------------------------------------------------------
//typedefs needed for the managed set interfaces
//-------------------------------------------------------------------
typedef sequence <Set>              SetList;
typedef sequence <Member>           MemberList;
typedef SysAdminTypes::ObjectLabel     MemberLabel;
typedef SysAdminTypes::ObjectLabelList MemberLabelList;

        interface Member : Identification::Labeled {

                void add_backref (
                        in Set s
                );

                void get_backrefs (
                        in unsigned long how_many,
                        out SetList s_list,
                        out SetIterator iterator
                );

                void remove_backref (
                        in Set s
                ) raises (
                        SysAdminException::ExNotFound
                );

        }; // End Member interface

        interface SetIterator {
```

```
            boolean next_one (
                    out Set s
            );

            boolean next_n (
                    in unsigned long how_many,
                    out SetList s_list
            );

            void destroy();

};  // End SetIterator interface


interface MemberIterator {

            boolean next_one (
                    out Member m
            );

            boolean next_n (
                    in unsigned long how_many,
                    out MemberList m_list
            );

            void destroy();

};  // End of MemberIterator

interface ObjectLabelIterator {

            boolean next_one (
                    out ObjectLabel ol
            );

            boolean next_n (
                    in unsigned long how_many,
                    out ObjectLabelList ol_list
            );

            void destroy();

};  // End of ObjectLabelIterator

typedef ObjectLabelIterator MemberLabelIterator;


interface Set : Member {

            unsigned long get_cardinality();

            void add_object (
                    in boolean add_backref,
                    in Member m
            ) raises (
                    SysAdminException::ExNotUniqueLabel
            );

            void add_n_objects (
```

```
                in MemberList m_list,
                out MemberList not_added
        );

        void remove_object (
                in boolean remove_backref,
                in Member m
        ) raises (
                SysAdminException::ExNotFound
        );

        void remove_n_objects (
                in boolean remove_backref,
                in MemberList m_list,
                out MemberList not_removed
        );

        void get_members (
                in unsigned long how_many,
                out MemberList m_list,
                out MemberIterator iterator
        );

        void intersection_members (
                in unsigned long how_many,
                in SetList s_list,
                out MemberList m_list,
                out MemberIterator iterator
        ) raises (
                SysAdminException::ExInvalid
        );

        void union_members (
                in unsigned long how_many,
                in SetList s_list,
                out MemberList m_list,
                out MemberIterator iterator
        ) raises (
                SysAdminException::ExInvalid
        );

}; // End of Set interface

interface FilteredSet : Set {

        void find_members (
                in SysAdminTypes::InterfaceDefList interfaces,
                in SysAdminTypes::LabelExpression regular_expression,
                in unsigned long how_many,
                out MemberLabelList ml_list,
                out MemberLabelIterator iterator
        ) raises (
                SysAdminException::ExNotFound
        );

        Member lookup_object (
                in SysAdminTypes::LabelType label,
                in SysAdminTypes::InterfaceDefList interfaces
        ) raises (
```

```
                    SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList lookup_labels (
                    in MemberList m_list
                );

        }; // End of FilteredSet interface

}; // End ManagedSets module

#endif //MANAGEDSETS_IDL
```

A.4 ManagedInstances.idl

```
//
// Component Name: ManagedInstances.idl
//
// Description:
// The following interfaces provide the functionality
// for support of instances and instance managers.
//

#ifndef MANAGEDINSTANCES_IDL
#define MANAGEDINSTANCES_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <SysAdminLifeCycle.idl>
#include <CosLifeCycle.idl>
#include <Policies.idl>
#include <ManagedSets.idl>


module ManagedInstances {

        // Forward reference
        interface Instance;
        interface BasicInstanceManager;
        interface InstanceManager;
        interface Library;

        interface Instance :
                CosLifeCycle::LifeCycleObject,
                ManagedSets::Member
        {

                BasicInstanceManager get_manager();

                string get_type_name();

                SysAdminLifeCycle::Location get_resource_location();

        }; // End of Instance interface

        interface Library :
                CosLifeCycle::FactoryFinder,
                CosLifeCycle::GenericFactory,
                ManagedSets::FilteredSet
        {
        }; // End of Library interface

        interface BasicInstanceManager :
                CosLifeCycle::LifeCycleObject,
                CosLifeCycle::GenericFactory,
                ManagedSets::FilteredSet
        {

                CORBA::InterfaceDef get_instances_interface();

        }; // End of BasicInstanceManager interface
```

```
        interface InstanceManager :
                BasicInstanceManager,
                Policies::PolicyObjectAdmin

        {
        }; // End of InstanceManager interface

        interface PolicyRegionsInstanceManager :
                InstanceManager
        {
        }; // End of PolicyRegionsInstanceManager interface

}; // End ManagedInstances module

#endif // MANAGEDINSTANCES_IDL
```

A.5 PolicyRegions.idl

```
//
// Component Name: PolicyRegions.idl
//
// Description:
// The following interfaces provide the functionality
// for support of policy regions.
//

#ifndef POLICYREGIONS_IDL
#define POLICYREGIONS_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <SysAdminLifeCycle.idl>
#include <CosLifeCycle.idl>
#include <ManagedSets.idl>
#include <ManagedInstances.idl>

module PolicyRegions {

// forward references
        interface PolicyRegion;
        interface PolicyDrivenBase;

        struct PolicyResult {
                PolicyDrivenBase object_verified;
                PolicyRegion containing_region;
                Policies::ValidationPolicy validation_object_used;
                boolean passed_policy;
                string description;
        };

        typedef sequence<PolicyResult> PolicyResultList;

        enum SelectionCriteria {
                all,
                with_initialization,
                with_validation,
                with_validation_enabled,
                with_initialization_or_validation,
                with_initialization_or_validation_enabled
        };

        typedef sequence<PolicyRegion> PolicyRegionList;

        interface PolicyResultIterator {
                boolean next_one (
                                out PolicyResult pr;
                };

                boolean next_n (
                        in unsigned long how_many,
                        out PolicyResultList pr_list;
                };

                void destroy();
```

```
};  // End PolicyResultIterator interface

interface PolicyDrivenBase : ManagedInstances::Instance {

        SysAdminTypes::ObjectLabelList get_policy_region_info();

        void move_to_policy_region (
                in PolicyRegions::PolicyRegion pr_from,
                in PolicyRegions::PolicyRegion pr_to
        ) raises (
                SysAdminException::ExObjNotFound,
                SysAdminException::ExNotFound,
                SysAdminException::ExInvalid
        );

        void add_to_policy_region (
                in PolicyRegions::PolicyRegion pr
        ) raises (
                SysAdminException::ExNotFound,
                SysAdminException::ExInvalid
        );

        void remove_from_policy_region (
                in PolicyRegions::PolicyRegion pr
        ) raises (
                SysAdminException::ExObjNotFound,
                SysAdminException::ExInvalid
        );

        SysAdminTypes::ObjectLabelList list_enabled_validation_policies (
                in PolicyRegionList policy_regions,
                in CORBA::InterfaceDef interface_def,
                in boolean include_nested
        ) raises (
                SysAdminException::ExNotFound
        );

        SysAdminTypes::ObjectLabelList list_initialization_policies
(
                in PolicyRegionList policy_regions,
                in CORBA::InterfaceDef interface_def,
                in boolean include_nested
        ) raises (
                SysAdminException::ExNotFound
        );

};  // End of PolicyDrivenBase interface

interface PolicyRegion :
        PolicyDrivenBase,
        ManagedSets::FilteredSet,
        CosLifeCycle::GenericFactory
{

        void add_instance_manager (
                in ManagedInstances::InstanceManager im,
                in Policies::InitializationPolicy initialization_policy,
                in Policies::ValidationPolicy validation_policy
        ) raises (
```

```
            SysAdminException::ExExists
);

void remove_instance_manager (
            in ManagedInstances::InstanceManager im
) raises (
            SysAdminException::ExObjNotFound,
            SysAdminException::ExExists
);

void get_instance_manager_list (
            in SelectionCriteria select
            in unsigned long how_many;
            out SysAdminTypes::ObjectLabelList ol_list;
            out ManagedSets::ObjectLabelIterator iterator;
);

void set_initialization_policy (
            in ManagedInstances::InstanceManager im,
            in Policies::InitializationPolicy initialization_policy
) raises (
            SysAdminException::ExObjNotFound
);

SysAdminTypes::ObjectLabel get_initialization_policy (
            in ManagedInstances::InstanceManager im
) raises (
            SysAdminException::ExObjNotFound
);

void set_validation_policy (
            in ManagedInstances::InstanceManager im,
            in Policies::ValidationPolicy validation_policy
) raises (
            SysAdminException::ExObjNotFound
);

SysAdminTypes::ObjectLabel get_validation_policy (
            in ManagedInstances::InstanceManager im
) raises (
            SysAdminException::ExObjNotFound
);

void policy_validation (
            in ManagedInstances::InstanceManager im,
            in boolean enable
) raises (
            SysAdminException::ExObjNotFound
);

boolean is_validation_enabled (
            in ManagedInstances::InstanceManager im
) raises (
            SysAdminException::ExObjNotFound
);

void verify_policy (
            in ManagedSets::Set scope,
            in boolean included_nested,
```

```
                        in unsigned long how_many,
                        out PolicyResultList pr_list,
                        out PolicyResultIterator iterator
                );

                PolicyResultList get_policy_failures (
                        in ManagedSets::Set scope,
                        in boolean include_nested
                        in unsigned long how_many,
                        out PolicyResultList pr_list,
                        out PolicyResultIterator iterator
                );

                SysAdminTypes::ObjectLabelList get_all_initialization_policies
(
                        in ManagedInstances::InstanceManager im
                ) raises (
                        SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList get_all_enabled_validation_policies (
                        in ManagedInstances::InstanceManager im
                ) raises (
                        SysAdminException::ExNotFound
                );

        }; // End of PolicyRegion interface

}; // End of PolicyRegions module

#endif // POLICYREGIONS_IDL
```

A.6 Policies.idl

```
//
// Component Name: Policies.idl
//
// Description:
// The following interfaces provide the functionality
// for support of policy objects.
//

#ifndef POLICIES_IDL
#define POLICIES_IDL

#include <SysAdminTypes.idl>
#include <SysAdminExcept.idl>
#include <Identification.idl>
#include <CosLifeCycle.idl>
#include <ManagedSets.idl>

module Policies {

        // forward references
        interface PolicyObject;
        interface PolicyObjectAdmin;
        interface InitializationPolicy;
        interface ValidationPolicy;

        interface PolicyObjectAdmin
        {

                SysAdminTypes::ObjectLabelList get_initialization_policies();

                InitializationPolicy get_default_initialization (
                ) raises (
                        SysAdminException::ExNotFound
                );

                SysAdminTypes::ObjectLabelList get_validation_policies();

                ValidationPolicy get_default_validation (
                ) raises (
                        SysAdminException::ExNotFound
                );

                void add_initialization (
                        in InitializationPolicy initialization_policy
                ) raises (
                        SysAdminException::ExExists,
                        SysAdminException::ExInvalid
                );

                void set_default_initialization (
                        in InitializationPolicy initialization_policy
                ) raises (
                        SysAdminException::ExInvalid
                );

                void remove_initialization (
                        in InitializationPolicy initialization_policy
```

```
                ) raises (
                        SysAdminException::ExObjNotFound
                );

                void add_validation (
                        in ValidationPolicy validation_policy
                ) raises (
                        SysAdminException::ExExists,
                        SysAdminException::ExInvalid
                );

                void remove_validation (
                        in ValidationPolicy validation_policy
                ) raises (
                        SysAdminException::ExObjNotFound
                );

                void set_default_validation (
                        in ValidationPolicy validation_policy
                ) raises (
                        SysAdminException::ExInvalid
                );

                void add_pr_backref (
                        in PolicyRegion pr
                );

                void remove_pr_backref (
                        in PolicyRegion pr
                ) raises (
                        SysAdminException::ExNotFound
                );

                void get_pr_backrefs (
                        in unsigned long how_many,
                        out SetList s_list,
                        out SetIterator iterator
                );

        }; // End of PolicyObjectAdmin interface

        // The PolicyObject interface defines the common
        // requirements of policy objects

        interface PolicyObject :
                Identification::Labeled
        {
                string get_policy_driven_object_type();

                CORBA::InterfaceDef get_policy_driven_object_interface();

        }; // End of PolicyObject interface

        // The InitializationPolicy interface defines the
        // requirements of an initialization policy object

        interface InitializationPolicy :
                PolicyObject
        {
```

```
                void initialize_policy_driven_object (
                        in Object object_to_initialize,
                        in CosLifeCycle::Criteria override_criteria
                );

        }; // End of InitializationPolicy interface

        // The ValidationPolicy interface defines the
        // requirements of an validation policy object

        interface ValidationPolicy :
                PolicyObject
        {

                boolean validate_policy_driven_object (
                        in Object object_to_validate,
                        out string description
                );

        }; // End of ValidationPolicy interface

}; // End of Policies module

#endif // POLICIES_IDL
```

A.7 SysAdminExcept.idl

```
//
// Component Name: SysAdminExcept.idl
//
// Description:
// The SysAdminExcept module defines the exceptions
// commonly used by system management applications.
//

#ifndef SYSADMINEXCEPT_IDL
#define SYSADMINEXCEPT_IDL

#include <SysAdminTypes.idl>

typedef sequence<any> MsgContext;

#define XPG_FIELDS                          \
        string    type_name;                \
        string    catalog;          \
        long      key;                      \
        string    default_msg;              \
        long      time_stamp;               \
        MsgContext      msg_context;

module SysAdminException {
        exception ExException {
                XPG_FIELDS

        };

        exception ExFailed {
                XPG_FIELDS
                string    operation_name;
        };

        exception ExInvalid {
                XPG_FIELDS
                string    resource_name;
        };

        exception ExNotUniqueLabel {
        XPG_FIELDS
        SysAdminTypes::LabelType        label;
        };

        exception ExNotFound {
                XPG_FIELDS
                string    resource_name;
        };

        exception ExExists {
                XPG_FIELDS
                string    resource_name;
        };

        exception ExObjNotFound {
                XPG_FIELDS
                string    resource_name;
```

```
        };
};

#endif //SYSADMINEXCEPT_IDL
```

A.8 SysAdminLifeCycle.idl

```
//
// Component Name: SysAdminLifecycle.idl
//
//
// Description:
// The SysAdminLifeCycle module defines interfaces
// for specifying objects. Currently interfaces for copying and
// moving objects are not supported. These will be added as needed.
//

#ifndef SYSADMINLIFECYCLE_IDL
#define SYSADMINLIFECYCLE_IDL

#include <SysAdminTypes.idl>

module SysAdminLifeCycle {

        // The Location interface allows the specification of
        // the location for lifecycle operations to occur.

        interface Location {
        };

        // the HostLocation interface allows the specification
        // of a particular client machine on which the
        // lifecycle operation should execute

        interface HostLocation : Location {

                SysAdminTypes::Platform get_platform();

        };

};

#endif // SYSADMINLIFECYCLE_IDL
```

Appendix B

Inheritance Relationships

This Appendix provides an inheritance diagram for the Common Management Facilities described in this specification.

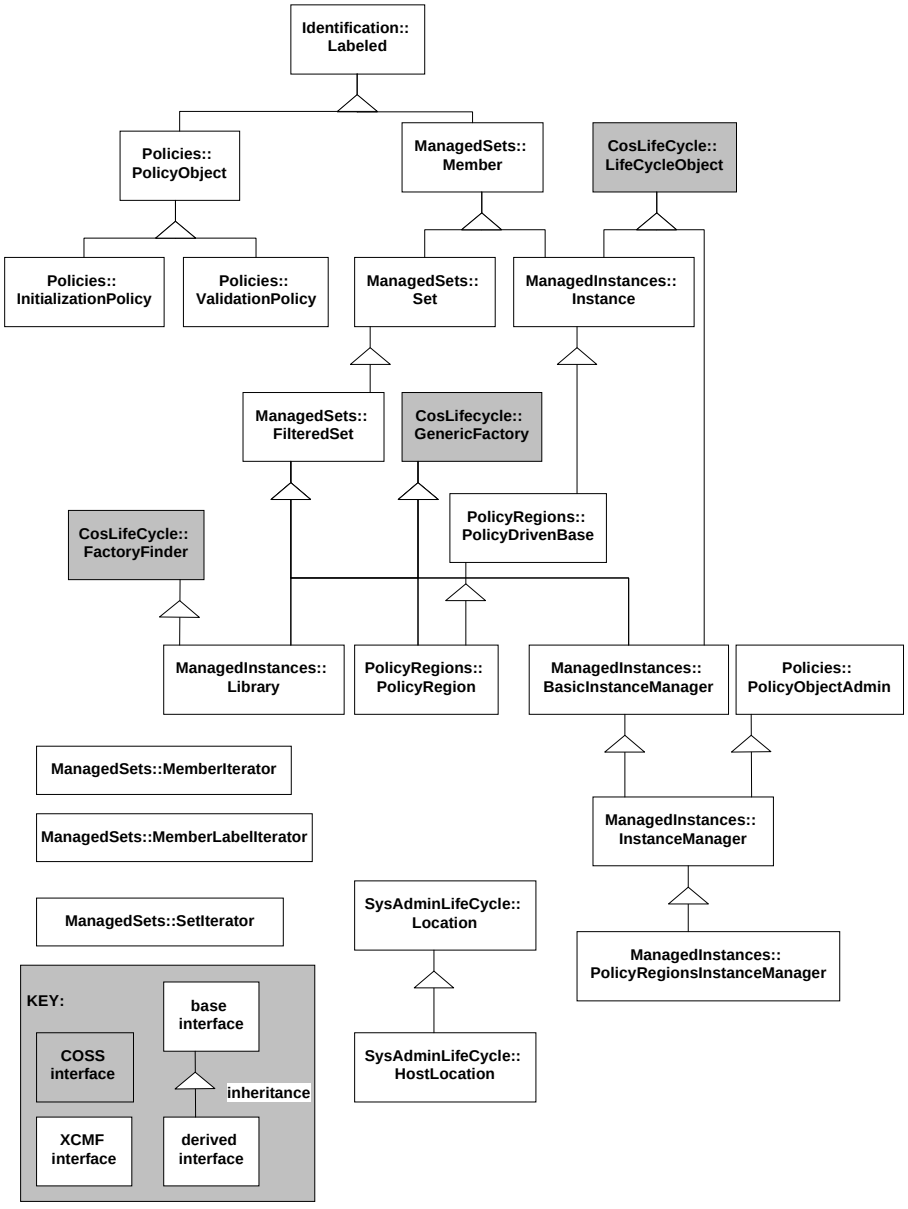


Figure B-1 Common Management Facilities: Inheritance

Glossary

activation
Preparing an object to execute an operation. For example, copying the persistent form of methods and stored data into an executable address space to allow execution of the methods on the stored data.

adapter
Same as object adapter.

attribute
An identifiable association between an object and a value. An attribute **A** is made visible to clients as a pair of operations: **get_A** and **set_A**. Read-only attributes only generate a **get** operation.

behaviour
The observable effects of an object performing the requested operation including its results.

client
The code or process that invokes an operation on an object.

context
A collection of name-value pairs that provides environmental or user-preference information.

CORBA
Common Object Request Broker Architecture.

COS
Common Object Services. See reference **COS Volume 1**.

COSS
Common Object Services Specification. See reference **COS Volume 1**.

data type
A categorization of values operation arguments, typically covering both behaviour and representation (that is, the traditional non-object-oriented programming language notion of type).

dynamic invocation
Constructing and issuing a request whose signature is possibly not known until run time.

event
A state change of an object that causes the behaviour of an object.

factory object
An object that creates another object.

federation
The principle whereby each component retains its autonomy rather than becoming subordinate to another.

implementation definition language
A notation for describing implementations. The implementation definition language is currently beyond the scope of the ORB standard. It may contain vendor-specific and adapter-specific notations.

implementation inheritance
The construction of an implementation by incremental modification of other implementations. The ORB does not provide implementation inheritance. Implementation inheritance may be provided by higher level tools.

inheritance
The construction of a definition by incremental modification of other definitions. See interface and implementation inheritance.

instance
An object is an instance of an interface if it provides the operations, signatures and semantics specified by that interface. An object is an instance of an implementation if its behaviour is provided by that implementation.

interface
A listing of the operations and attributes that an object provides. This includes the signatures of the operations, and the types of the attributes. An interface definition ideally includes the semantics as well. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface.

interface inheritance
The construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.

interface object
An object that serves to describe an interface. Interface objects reside in an interface repository.

interface type
A type satisfied by any object that satisfies a particular interface.

interoperability
The ability for two or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.

language binding
The means and conventions by which a programmer writing in a specific programming language accesses ORB capabilities.

language mapping
Language binding.

life cycle object
An object whose interfaces are defined by the life cycle services, specifically remove, copy and move.

method
An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.

multiple inheritance
The construction of a definition by incremental modification of more than one other definition.

name binding
A name-to-object association. A name binding is always defined relative to a naming context.

object
A combination of state and a set of methods that explicitly embodies an abstraction characterized by the behaviour of relevant requests. An object is an instance of an implementation and an interface. An object models a real-world entity, and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requestor services.

object reference
A value that unambiguously identifies an object. Object references are never reused to identify another object.

OMA
Object Management Architecture

OMG
Object Management Group

operation
A service that can be requested. An operation has an associated signature, which may restrict which actual parameters are valid.

ORB
Object Request Broker. Provides the means by which clients make and receive requests and responses. A persistent object exists until it is explicitly deleted.

relationship
Relationships allow semantics to be added to references between objects. For example, relationships allow one object to contain another. Life cycle services must work in the presence of graphs of related objects.

request
A client issues a request to cause a service to be performed. A request consists of an operation and zero or more actual parameters.

results
The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

server
A process implementing one or more operations on one or more objects.

signature
Defines the parameters of a given operation including their number order, data types and passing mode; the results if any; and the possible outcome (normal as opposed to exceptional) that might occur.

single inheritance
The construction of a definition by incremental modification of one definition. Contrast with multiple inheritance.

state
The time varying properties of an object that affect its behaviour.

stub
A local procedure corresponding to a single operation that invokes that operation when called.

typed event
An event for which an interface is defined in terms of IDL.

value
Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.

Index