Online Upgrades

Draft Adopted Specification

Eternal Systems, Inc.
Objective Interface Systems, Inc.
Vertel Corporation

With support and collaboration from:
Cacheon, Inc.
University of California, Santa Barbara

OMG Document ptc/2003-08-07
Amended with change bars by the finalization task force
August16, 2003

Table of Contents

1. Introduction 3
1.1 Submitting Companies 3
1.2 Status of the Document 3
1.3 Guide to the Submission 3
1.4 Proof of Concept 4
1.5 Submission Contact Points 5

2. Design Rationale 6
2.1 Motivation 6
2.2 Objectives 7
2.3 Limitations 7
2.4 Overview of the Online Upgrade Specifications 8
2.5 Successive Stages of an Upgrade 9
2.6 Object References and Message Forwarding 12
2.7 Online Upgrade Scenarios 14
2.8 Extensibility of the Specification 16
2.9 Platform Independent Model and a Mapping to CORBA 17

3. Group Management 26
3.1 Overview 26
3.2 PortableGroup Module 27
3.3 Properties of Upgradeable Objects 46
3.4 GroupManager Interface 47

4. Upgrade Management 53
4.1 Overview 53
4.2 UpgradeManager Interface 53

5. Upgradeable Applications 60
5.1 Overview 60
5.2 PortableState Module 61
5.3 Upgradeable Interface 63

6. Usage of the Specifications 69
6.1 Example Use Case 69

7. Responses to RFP Requirements 76
7.1 Resolution of RFP Mandatory Requirements 76
7.2 Resolution of RFP Optional Requirements 78
7.3 Responses to RFP Issues 80

8. Compliance and Conformance 81
8.1 Mandatory and Optional Interfaces 81
8.2 Proposed Compliance Points 81
8.3 Changes to Existing Specifications 81

9. Consolidated IDL 82

Introduction 1

1.1 Submitting Companies

The following companies are pleased to submit this specification in response to the Online Upgrades RFP (Document orbos/2001-09-10).

Eternal Systems, Inc.

Objective Interface Systems, Inc.

Vertel Corporation

The following organizations are pleased to support this proposal.

Cacheon, Inc.

University of California, Santa Barbara

1.2 Status of the Document

This document is a revised submission produced for the OMG Technical Committee meeting to be held in Orlando, Florida, in June 2002.

1.3 Guide to the Submission

This submission presents specifications for CORBA IDL interfaces and also a Platform Independent Model to support Online Upgrades of CORBA object implementations.

Chapter 1 contains the Introduction to this Online Upgrades proposal, and Chapter 2 provides the Design Rationale. Chapter 3 defines the specification of the **PortableGroup** module and the **GroupManager** interface, Chapter 4 defines the specification of the **UpgradeManager** interface, and Chapter 5 defines the **PortableState** module and the **Upgradeable** interface that an application object that is to be upgraded must inherit. Chapter 6 provides an Example Use Case of the specification. Chapter 7 contains the Responses to the RFP Requirements and Issues, and Chapter 8 provides the Compliance and Conformance points of the proposal. The Appendix contains the Consolidated IDL for the proposed specification.

1.4 Proof of Concept

The proposed specifications is based, in part, on prototype implementations developed at Eternal Systems, Inc., and at the University of California, Santa Barbara.

1.5 Submission Contact Points

David Stringer
Cacheon, Inc.
785 Market Street, 10th Floor
San Fransciso, CA 94103
phone: +1 415 777-2555
fax: +1 415 777-5666
email: david.stringer@cacheon.com

Louise Moser
Eternal Systems, Inc.
5290 Overpass Road, Building D
Santa Barbara, CA 93111
phone: +1 805 448-8249
fax: +1 805 696-9083
email: moser@eternal-systems.com

Bill Beckwith
Objective Interface Systems, Inc.
13873 Park Center Road, Suite 360
Herndon, VA 20171
phone: +1 703 295-6500
fax: +1 703 295-6501
email: bill.beckwith@ois.com

Shahzad Aslam-Mir
Vertel Corporation
5741 Pacific Center Boulevard
San Diego, CA 92121
phone: +1 858 824-4128
fax: +1 858 824-4110
email: sam-aslam-mir@vertel.com

Michael Melliar-Smith
University of California, Santa Barbara
Department of Electrical and Computer Engineering
Santa Barbara, CA 93106
phone: +1 805 448-8250
fax: +1 805 893-3262
email: pmms@ece.ucsb.edu

Design Rationale 2

2.1 Motivation

Many large complex systems, and also many embedded systems, are required to provide continuous service to their users without interruption or suspension of service. To achieve continuous service, despite growth and evolution, it must be possible to upgrade such a system by replacing individual software and hardware components with new and upgraded components. Many systems cannot be taken out of service to perform an upgrade, and it is often difficult to take part of a system out of service for upgrading while other parts of the system continue to operate.

In the current state-of-the-art, most systems are halted for upgrading, resulting in a loss of service. Even when upgrades are performed while the system continues to provide service, the current practice is poor. Existing mechanisms are proprietary, difficult to use, prone to fiascos, and certainly not portable or interoperable.

Global financial systems, operating 24 X 7, are examples of computer systems that cannot be shut down for upgrading. Automated global supply chain applications have similar characteristics. Loss of service provided by such a system, whether unintentionally caused by a fault or deliberately caused to facilitate an upgrade, is undesirable because it disrupts not only the operation of that system but also the computer systems of many customers, suppliers and partners. Moreover, the rate at which changes and enhancements are introduced into such systems is accelerating, increasing the frequency with which service must be suspended for upgrades.

Embedded elecommunications, transportation, industrial control, defense and aerospace applications must also operate continuously for long periods of time, and must be upgraded without suspension of service. Portability and interoperability are important for embedded systems because many of them are built from subsystems supplied by different vendors. Existing proprietary upgrade strategies might work adequately in a single-vendor system, but can disrupt the operation of other subsystems in a multi-vendor system. Consequently, a common standard for online upgrades, shared by all vendors, is important.

2.2 Objectives

The main objective of this proposed specification for Online Upgrades is to facilitate the safe and orderly upgrading of objects in a manner that is portable across systems and that is interoperable between systems.

The proposed specification for Online Upgrades is a first step towards a more general online upgrade capability. The specification aims to provide the ability to:

- Upgrade individual objects, where such upgrades change the implementation of the object but do not change the external interfaces of the object
- Pause an object, so that it can be upgraded, while allowing the object the opportunity to reach a safe and quiescent state
- Transfer state from an instance of the old implementation of the object to an instance of the new implementation of the object, with provision for such state transfers where the representations of the old state and the new state are different
- Resume service using an instance of the new implementation of the object without risk that messages will be lost, misordered or processed twice
- Allow client objects to continue to use a server object while remaining unaware that the server has been upgraded, and allow server objects to continue to serve a middle-tier client object that also acts as a server while remaining unaware that the client has been upgraded
- Address objects in such a way that a client can continue to use its existing object reference to access a server after it has been upgraded
- Rollback an upgrade, prior to the instance of the new implementation becoming operational, if some part of the upgrade fails
- Revert from an instance of the new implementation to an instance of the old implementation, if operation with the instance of the new implementation proves to be unsatisfactory
- Perform upgrades on small collections of objects by means of allowing the application to commit and rollback the upgrades explicitly.

2.3 Limitations

The proposed specification for Online Upgrades does not provide interfaces or mechanisms to:

- Upgrade external interfaces of an object
- Allow an object to initiate its own upgrading
- Operate instances of both the old implementation and the new implementation concurrently
- Revert to an instance of a prior implementation other than an instance of the immediately preceding implementation
- Test new implementations

- Define version numbers for implementations
- Determine when an upgrade is available and when it should be applied to an object
- Allow the user to modify the values of variables of the instance of the upgraded implementation (which might be corrupt) when transforming the state and reverting to the old implementation of the object
- Determine the security or validity of an upgrade.

It is envisaged that the above services would be provided by higher level application software that would exploit the Online Upgrade infrastructure to perform the upgrading of individual objects. It is also envisaged that additional CORBA specifications might be developed to define some of these higher level capabilities.

2.4 Overview of the Online Upgrade Specifications

This proposal defines an **OnlineUpgrades** module that contains two interfaces: the **UpgradeManager** interface and the **GroupManager** interface, as shown in Figure 2-1. The proposal also defines a **PortableGroup** module that contains the **PropertyManager**, **ObjectGroupManager** and **GenericFactory** interfaces, which the **GroupManager** interface inherits. It also defines a **PortableState** module that contains the **Checkpointable** interface.

The **UpgradeManager** interface, which the Upgrade Manager object implements, is the principal interface that a user, or higher level management software, uses to achieve an online upgrade. The user, or higher level management software, invokes the methods of the Upgrade Manager to initiate, control, commit and revert online upgrades.

The **GroupManager** interface, which the Group Manager object implements, provides a significant part of the functionality of the online upgrade architecture. Alternatively, the Upgrade Manager, or another object such as the Replication Manager of a Fault Tolerant CORBA implementation, might inherit and implement the interfaces of the **PortableGroup** module. The methods of the Group Manager are invoked by the Upgrade Manager, or by a user who is exercising precise application control over group management.

An object that is to be upgraded must inherit the **Upgradeable** interface. The methods of this interface concern reporting that an object instance is in a safe and quiescent state so that it can be paused for upgrading, and for transferring the state from an instance of the old implementation of the object to an instance of the new implementation of the object. The methods are invoked by the Upgrade Manager.

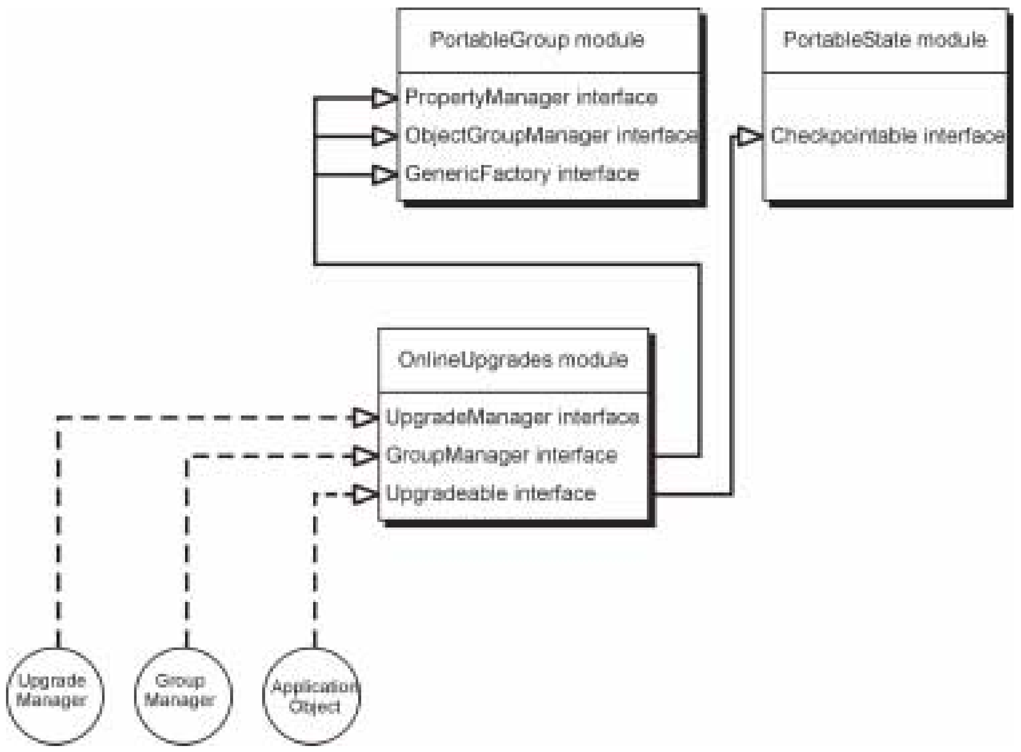


Figure 2-1 Online upgrade modules, interfaces and objects

The methods of the **Upgradeable** interface are programmed by the application programmer, and must be programmed for each class of objects, because they depend on the internal data structures of the particular class of objects. For simple classes, a vendor might provide a preprocessor that takes, as input, the source code of the class and generates, as output, source code for the methods of this interface.

2.5 Successive Stages of an Upgrade

Figure 2-2 shows the successive stages of an upgrade. In Stage 1, an instance of the old implementation of the object is operating. The object must inherit the **Upgradeable** interface but, otherwise, has no special characterisitics. We assume that the instance of the object-to-be-upgraded is not already a member of an object group, as indicated by its being addressed by an Interoperable Reference (IOR), rather than an Interoperable Group Reference (IOGR). If the object-to-be-upgraded is already a member of an object group, which is addressed by an IOGR, the existing object group containing that member is used and there is no need to create a new group.

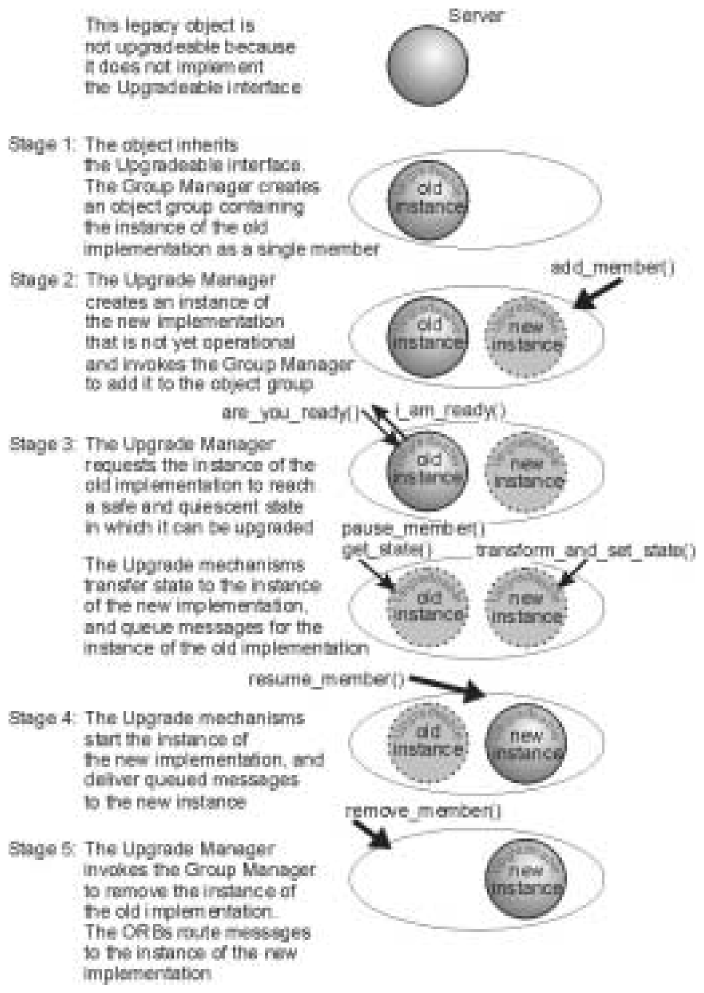

This legacy object is not upgradeable because it does not implement the Upgradeable interface
Server
Stage 1: The object inherits the Upgradeable interface. The Group Manager creates an object group containing the instance of the old implementation as a single member
old instance
Stage 2: The Upgrade Manager creates an instance of the new implementation that is not yet operational and invokes the Group Manager to add it to the object group
add_member()
new instance
are_you_ready()
i_am_ready()
Stage 3: The Upgrade Manager requests the instance of the old implementation to reach a safe and quiescent state in which it can be upgraded
pause_member()
get_state()
transform_and_set_state()
The Upgrade mechanisms transfer state to the instance of the new implementation, and queue messages for the instance of the old implementation
resume_member()
Stage 4: The Upgrade mechanisms start the instance of the new implementation, and deliver queued messages to the new instance
remove_member()
Stage 5: The Upgrade Manager invokes the Group Manager to remove the instance of the old implementation. The ORBs route messages to the instance of the new implementation

Figure 2-2 Successive Stages of an Upgrade

In Stage 2, first we check that the object inherits the **Upgradeable** interface. If the membership style is infrastructure controlled, the higher level application software invokes the **upgrade_object()** method of the Upgrade Manager. The Upgrade Manager then forms an object group with no members, using the **create_object()** method of the **GenericFactory** interface of the Group Manager. The Upgrade Manager then includes the existing object in that group using the **add_member()** method of the Group Manager. Next, the Upgrade Manager creates an instance of the new implementation of the object, using the **create_member()** method of the Group Manager, checking that the IDL interface of the new implementation is identical to that of the old implementation. The Upgrade Mechanisms (which are vendor-specific) log messages for the new member of the group, but do not yet deliver messages to it.

If the MembershipStyle is application controlled, the higher level application software forms an object group with no members, using the **create_object()** method of the **GenericFactory** interface of the Group Manager. The higher level application software then includes the existing object in the group using the **add_member()** method of the

Group Manager. Next, the higher level application software creates an instance of the new implementation of the object, using the **create_member()** method of the Group Manager, checking that the IDL interface of the new implementation is identical to that of the old implementation. The higher level application software then invokes the **upgrade_object()** method of the Upgrade Manager. The Upgrade Mechanisms log messages for the new member of the group, but do not yet deliver messages to it.

In Stage 3 (regardless of which MembershipStyle is used), the Upgrade Manager queries the object being upgraded to determine whether it is in a safe and quiescent state by invoking the **are_you_ready()** method of the **Upgradeable** interface of an instance of the old implementation. A safe state may be determined by the internal state of the implementation and, possibly, by the state of other objects or of physical equipment being controlled by the object. A quiescent state is a state in which the object is not executing any method that has been invoked on it.

The Upgrade Mechanisms do not deliver any further method invocations to the instance of the old implementation (except for the invocation of **get_state()** described below); instead, they queue all such request messages and deliver them in due course to the instance of the new implementation of the object (assuming that the QuiescenceStyle property defined in Section 3.7 has the value false).

When the object is ready to be upgraded, i.e., is in a safe and quiescent state, it invokes the **i_am_ready()** method of the Upgrade Manager, with the **ready** parameter set to true, which allows the upgrade to proceed. If the object invokes the **i_am_ready()** method, with the **ready** parameter set to false, then the Upgrade Manager rolls back the upgrade. The instance of the old implementation may delay its reply until it has reached a safe and quiescent state.

If the object invokes **i_am_ready()** method, with the **ready** parameter set to true, the Upgrade Mechanisms invoke the **get_state()** method of the **Upgradeable::Checkpointable** interface of the instance of the old implementation. The instance of the new implementation is started by invoking the **transform_and_set_state()** method of the **Upgradeable** interface of the instance of the new implementation. The parameter of this method invocation is the state returned by the **get_state()** invocation of the instance of the old implementation transformed into the state of the instance of the new implementation, including appropriate values for new attributes, etc. No other request messages are delivered to the instance of the new implementation yet.

At this point, if the **upgrade_object()** method is invoked with a value of false for its **app_ctrl_commit** parameter, the Upgrade Manager proceeds directly to Stage 4. Alternatively, if the **upgrade_object()** method is invoked with a value of true for its **app_ctrl_commit** parameter, Stage 4 is entered when the application invokes the **commit_upgrade()** method of the Upgrade Manager.

The **app_ctrl_commit** parameter is used, when its value is set to true, to allow a collection of objects to be upgraded together. Initially, the **upgrade_object()** method is invoked to upgrade each object in the collection. Each such object is paused, but not committed because the **app_ctrl_commit** parameter is set to true. If all of the upgrades of the objects in the collection are successful, then the **commit_upgrade()** method is invoked for each object in the collection to commit the upgrade of that object. If any of the upgrades fails, then the **rollback_upgrade()** method is used to undo the upgrade for each of the objects that had been upgraded but not committed.

In Stage 4, the Upgrade Mechanisms now apply the queued messages to the instance of the new implementation, and direct all future messages to the instance of the new implementation. For infrastructure (application) controlled MembershipStyle, the Upgrade Manager (application level software) removes the instance of the old implementation from the object group by use of the **remove_member()** method of the Group Manager. The upgrade has now reached Stage 5 of Figure 2-2.

Alternatively, if the application level software invokes the **rollback_upgrade()** method of the Upgrade Manager, the instance of the old implementation resumes normal operation. For the infrastructure (application) controlled MembershipStyle, the Upgrade Manager (application level software) removes the instance of the new implementation from the object group by use of the **remove_member()** method of the Group Manager.

Implementers of this specification may take appropriate steps to ensure that a rollback can be accomplished whether requested explicitly, triggered implicitly by way of an exception condition during the upgrade process, or caused by system or network failure during the upgrade process. How recovery and rollback are facilitated is implementation-specific. For example, a copy of the message queue may be stored at a remote processor or may be persisted to disk (so that, in the event of a fault, the queue can be reconstituted).

2.6 Object References and Message Forwarding

The proposed Online Upgrade specification exploits the Interoperable Group Reference (IOGR), introduced in the Fault Tolerant CORBA standard and since used by the Unreliable Multicast, Data Parallel CORBA and Load Balancing specifications.

The advantage of the IOGR is that it can contain multiple profiles, in particular the profile for the old implementation of an object and the profile for the new implementation of the object. The IOGR also allows the use of the LOCATION_FORWARD_PERMANENT reply, which provides a client with a new IOGR containing the address of the instance of the new implementation and directs a client to use that IOGR for all future invocations of the object.

If the instance of the new implementation of an object is colocated with the instance of the old implementation, i.e., in the same process and supported by the same POA, the upgrade can be performed without the use of the IOGR. However, the proposed specifications are intended for use where an instance of the new implementation is not colocated with an instance of the old implementation, such as when an upgrade is performed from old hardware to new hardware.

If the instance of the new implementation and the instance of the old implementation are not colocated, we consider two alternatives:

- The Group Manager is supported by a multicast protocol that delivers messages to all members of the group.

- The object being upgraded communicates with other objects via IIOP using point-to-point communication.

If the QuiescenceStyle property defined in Section 3.7 has the value false, when the Upgrade Manager invokes the **are_you_ready()** method, the Upgrade Mechanisms stop delivering request messages to the instance of the old implementation but, rather, start queuing such messages.

- If the instance of the old implementation and the instance of the new implementation are colocated, they share Upgrade Mechanisms that queue messages at that location.
- If the instance of the old implementation and the instance of the new implementation are not colocated and multicast group communication is used, the Upgrade Mechanisms multicast the messages to both instances and queue the request messages at both instances but deliver them to neither.
- If the instance of the old implementation and the instance of the new implementation are not colocated and IIOP is used, the Upgrade Mechanisms route messages to the instance of the old implementation and queue the request messages there, and do not route the messages to the instance of the new implementation yet.

Next, the Upgrade Manager switches over from the instance of the old implementation to the instance of the new implementation.

- If the instance of the old implementation and the instance of the new implementation are colocated, the Upgrade Mechanisms can deliver the queued messages directly to the instance of the new implementation.
- If multicast communication is used, the Upgrade Mechanisms have queued those messages at the instances of both implementations, and can deliver the messages to the instance of the new implementation.
- If the instance of the old implementation and the instance of the new implementation are not colocated and IIOP is used then, as shown in Figure 2-3, for each of the messages queued at the instance of the old implementation, the Upgrade Mechanisms return a LOCATION_FORWARD reply to the client that originated the queued message. The client ORB retransmits the message to the instance of the new implementation. For the last message queued from any one client, the reply contains a LOCATION_FORWARD_PERMANENT, instead of a LOCATION_FORWARD. The LOCATION_FORWARD_PERMANENT reply carries with it the new IOGR for the object group, which contains the profile of the new implementation of the object, and causes the ORB to update the client's object reference so that future messages are sent to the instance of the new implementation.

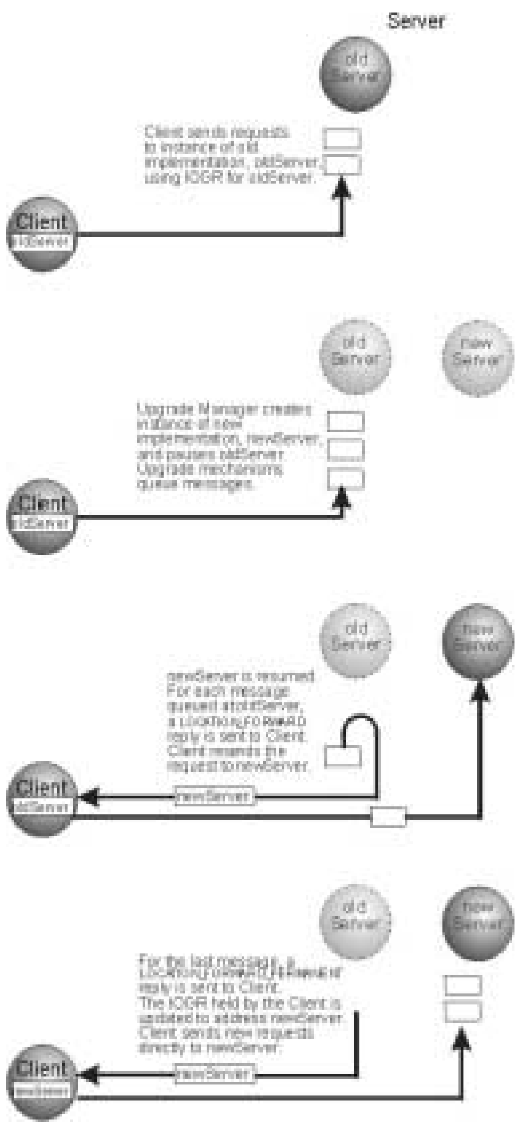

Server
old Server
Client sends requests to instance of old implementation, oldServer, using IOGR for oldServer.
Client
oldServer
old Server
new Server
Upgrade Manager creates instance of new implementation, newServer, and pauses oldServer. Upgrade mechanisms queue messages.
Client
oldServer
old Server
new Server
newServer is resumed. For each message queued at oldServer, a LOCATION_FORWARD reply is sent to Client. Client resends the request to newServer.
newServer
Client
oldServer
old Server
new Server
For the last message, a LOCATION_FORWARD_PERMANENT reply is sent to Client. The IOGR held by the Client is updated to address newServer. Client sends new requests directly to newServer.
newServer
Client
newServer

Figure 2-3 Object References and Message Forwarding

2.7 Online Upgrade Scenarios

Three upgrade scenarios are discussed below and shown in Figure 2-4.

2.7.1 Pushed Upgrade

In this scenario a selected server object is upgraded under the control of a distinct software system external to the object-to-be-upgraded. This distinct piece of software is responsible for driving the upgrade process and pushing the upgrade to the object-to-be-upgraded.

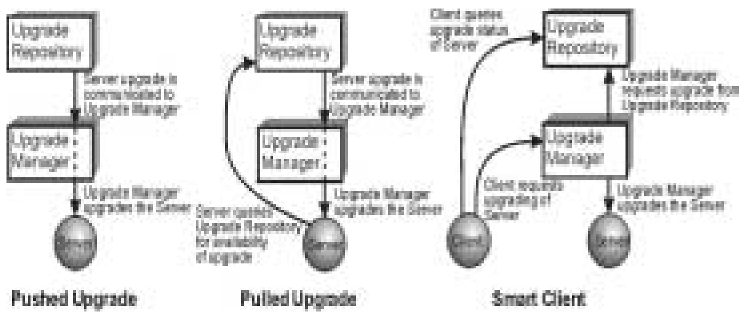

Upgrade Repository
Server upgrade is communicated to Upgrade Manager
Upgrade Manager
Upgrade Manager upgrades the Server
Server
Pushed Upgrade
Upgrade Repository
Server upgrade is communicated to Upgrade Manager
Upgrade Manager
Upgrade Manager upgrades the Server
Server queries Upgrade Repository for availability of upgrade
Server
Pulled Upgrade
Client queries upgrade status of Server
Upgrade Repository
Upgrade Manager requests upgrade from Upgrade Repository
Upgrade Manager
Client requests upgrading of Server
Upgrade Manager upgrades the Server
Client
Server
Smart Client

Figure 2-4 Online Upgrade Scenarios

The proposal supports this scenario outright with an Upgrade Manager (the distinct external subsystem) that is responsible for driving an upgrade process. It is expected that a third piece of software (part of the application) will command the Upgrade Manager to begin the upgrade process and push the upgrade.

This is not to say that the object-to-be-upgraded has no control in the process. In fact, the object-to-be-upgraded must accept a message sent by the Upgrade Manager requesting that it prepare itself for upgrade and respond when it is ready. The remainder of the process is then conducted by the Upgrade Manager.

2.7.2 Pulled Upgrade

In this scenario a server object-to-be-upgraded is responsible for initiating its own upgrade. For example, a server object may, when idle or at scheduled times, make a call to an external upgrade repository to determine if an upgrade is available for it or when such an upgrade is expected to be available. On successful discovery of an available upgrade, the object-to-be-upgraded can initiate the upgrade process, effectively pulling the upgrade to it.

The current proposal supports a pulled upgrade, but considers an upgrade repository to be orthogonal to the switchover process. If an upgrade repository is available, an object-to-be-upgraded can determine the availability of an upgrade as described above and then make a call to the Upgrade Manager to initiate the upgrade. From that point on, the process is identical to the pushed upgrade process. An object-to-be-upgraded can utilize the same set of interfaces of the Upgrade Manager to accomplish an upgrade as would any other application code that initiates an upgrade.

2.7.3 Upgrades in the Presence of Smart Clients and System Management

Clients that invoke methods of an object-to-be-upgraded are typically expected to be functionally unaware of the upgrade process, but this does not always hold. Advanced systems may make use of smart clients, which have more say in how their requests are handled than a typical client.

For example, a smart client might monitor the response time, possibly backing out of an interaction with a server object and choosing a different server object, if the response time is too long. In such a case, a smart client must be able to determine when one of its target server objects is being upgraded, monitor the progress of the upgrade, and potentially cancel its invocation of that server or redirecting its request to another server.

In addition, a system management program might have the need to monitor upgrade processes, potentially postponing or even killing an upgrade process process if it conflicts with other system and application management priorities.

The proposal supports both smart clients and system management by providing interfaces of the Upgrade Manager for the purposes of monitoring the upgrade process phase-by-phase and also supporting an upgrade with explicit commit and rollback commands.

2.8 Extensibility of the Specification

The proposed Online Upgrade specification is designed to provide a basic online upgrade capability, primarily the upgrading of an instance of the implementation of a single object with no change to the interface of the object. However, the specification is designed to provide a building block, on top of which a wide variety of more sophisticated software systems can be constructed.

For example, more sophisticated software upgrade systems, that are capable of upgrading object interfaces including method signatures, use a strategy of performing a complex upgrade as a sequence of simple upgrades, each of which is implemented by the basic upgrade operation defined in the proposed specification.

Fault Tolerant CORBA, with its objective of continuous service despite faults, might be enhanced with online upgrades to ensure that the continuous service objective is maintained even while the software is being upgraded.

Load balancing systems based on the proposed CORBA Load Balancing specifications are limited to stateless objects. Such systems could be augmented with the ability to migrate a stateful object by upgrading an object located on one processor to another functionally identical object located on a different processor.

Hardware and software configuration management software can use the basic online upgrade capability to integrate new software into a system, or to populate new hardware with already operating software, possibly adapting that software to the new platform on which it will operate.

Some of these capabilities may require an implementation that is at least partly integrated with the implementation of Online Upgrades. For example, providing Online Upgrades for Fault Tolerant CORBA will require mechanisms that can maintain the desired degree of replication during an upgrade. Other users of the PortableGroup interface may construct heterogeneous object groups, and the Upgrade Mechanisms must be able to select the appropriate member of such a group for upgrading.

2.9 Platform Independent Model and a Mapping to CORBA

It is our goal that the Online Upgrade process advocated in the current proposal be applicable to multiple technology platforms, including CORBA, Java, J2EE, .NET, EDOC, and others as appropriate. For example, the Java Community Process is concurrently developing a specification of Continuous Availability JSR-117, which aims to support online upgrades for EJB/J2EE enterprise applications. Likewise, the Service Availability Forum (SAF) is developing a specification of Service Availability for embedded applications, in particular telecommunications and data communications applications. We intend the process specified here, suitably abstracted as regards technology specifics, to be applicable to JSR-117 and the SAF specifications. To achieve this end, we embrace the notions introduced by the OMG's Model Driven Architecture (MDA). MDA introduced the concepts of PIM, PSM, and mapping, which we apply below.

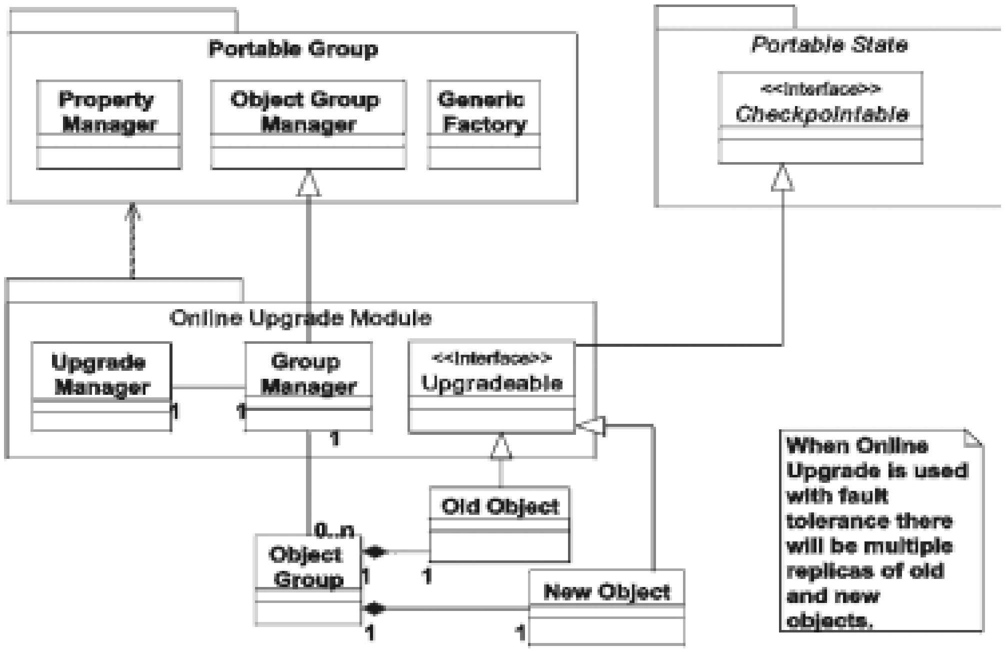
PIM, which stands for Platform Independent Model, is a representation of a system or application structure, behavior and function in a form that is independent of the concerns and details of the specific technology platform employed in an implementation, such as CORBA, Java, J2EE, or .NET.

PSM, standing for Platform Specific Model, is a representation of a system or application structure, behavior and function in a form that specifically addresses the concerns and details of a selected technology platform.

Both PIM and PSM approaches can address a variety of levels of abstraction, but a PIM is more abstract than a PSM (as it abstracts out platform-specific details). Interrelating models - across various levels of abstraction, between platform-independent and platform-specific, and across system roles and subsystems - is the job of inter-model mappings.

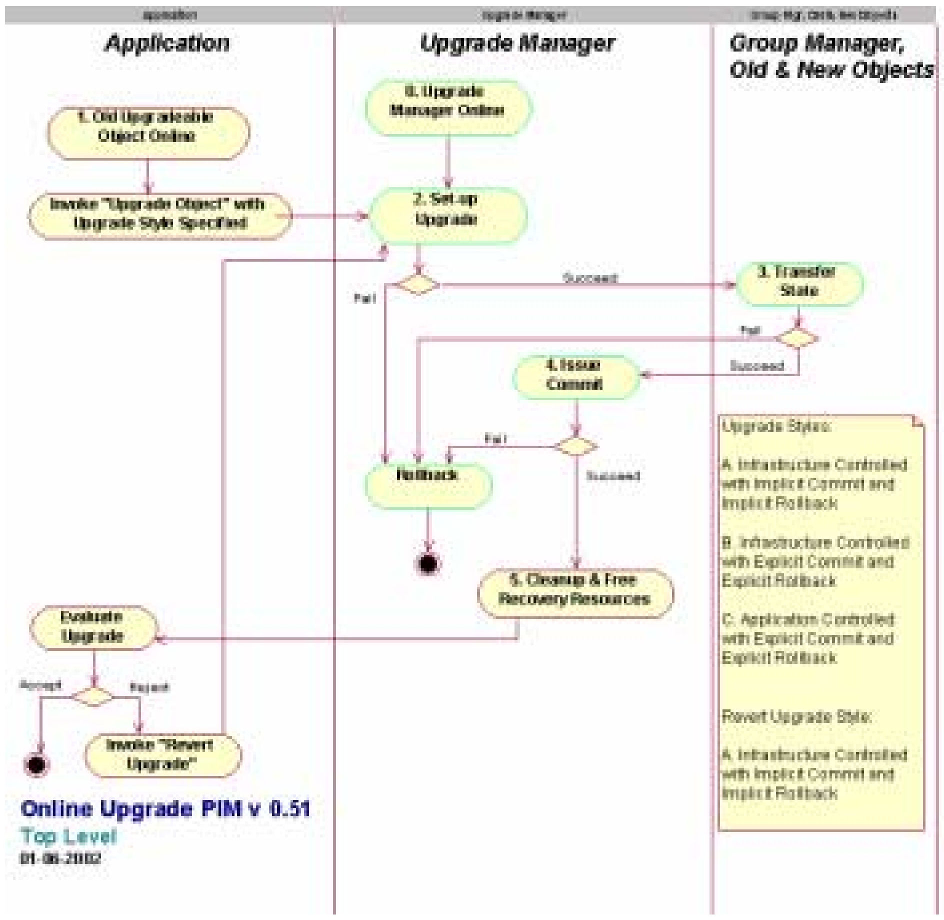
At the time of the current proposal, MDA is just over one year old (based on the public introduction of the MDA label), with the consequence that it is in an early formative stage. It is expected that PIMs, PSMs, and inter-model mappings will all be expressed using UML 1.x in the short term, ultimately yielding to the UML 2.0 work currently in process.

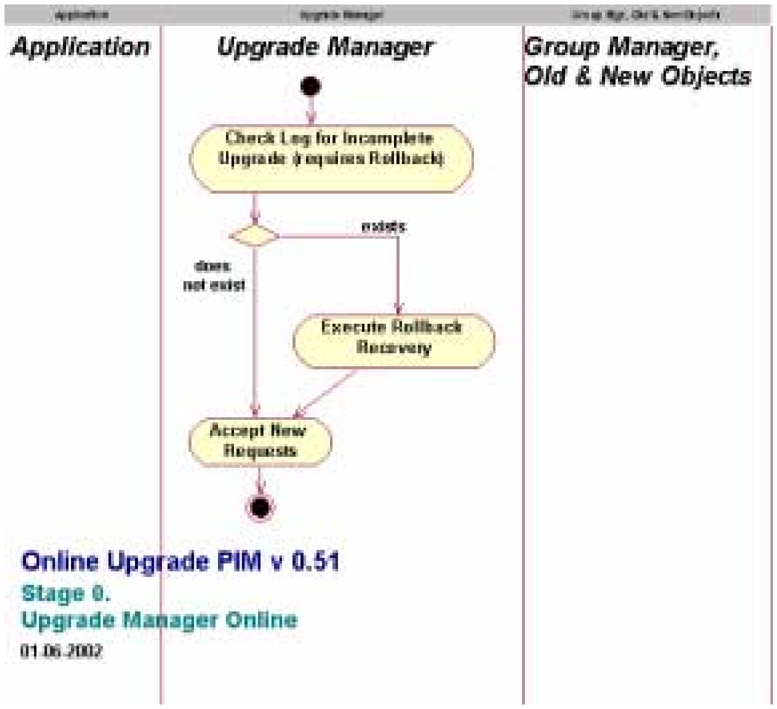
As shown in Figure 2-5, the PIM consists of one Class diagram and seven Activity diagrams, one of which serves as a master that ranges over all five stages of the upgrade process and six that serve to represent the stages in greater detail. The class diagram represents the major responsible parties and subsystems involved in an Online Upgrade and their associations. The multiplicities shown in this class diagram are normative. The Activity diagrams use swim lanes to segregate activities by the responsible party or subsystem.

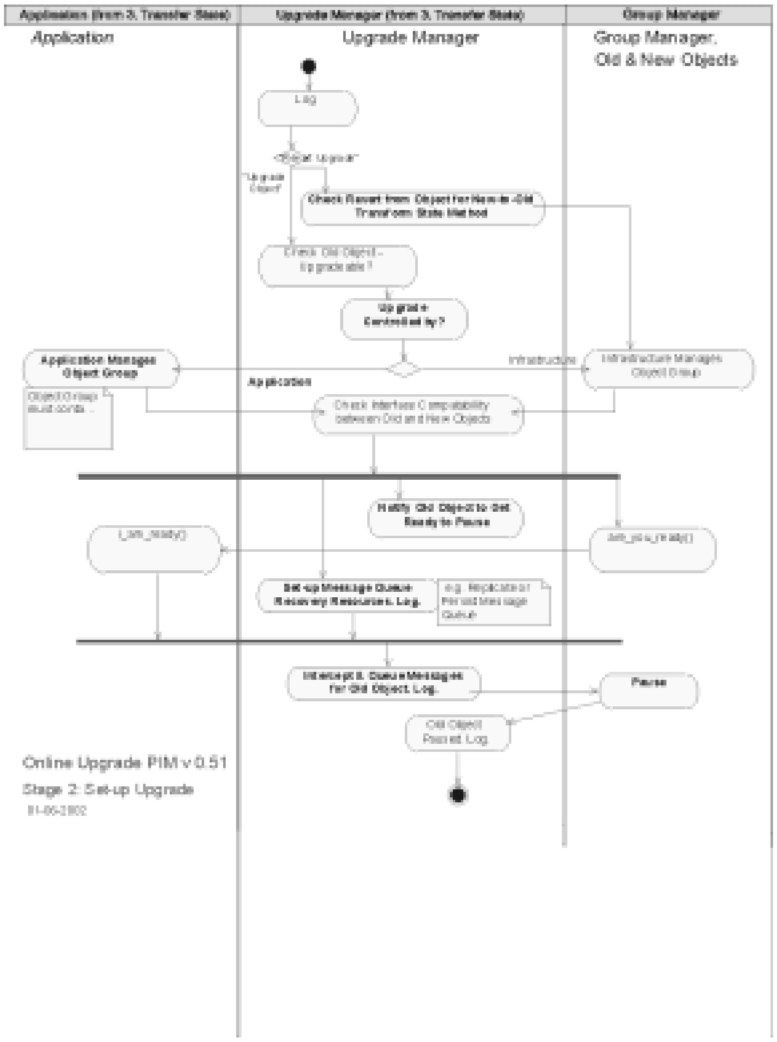


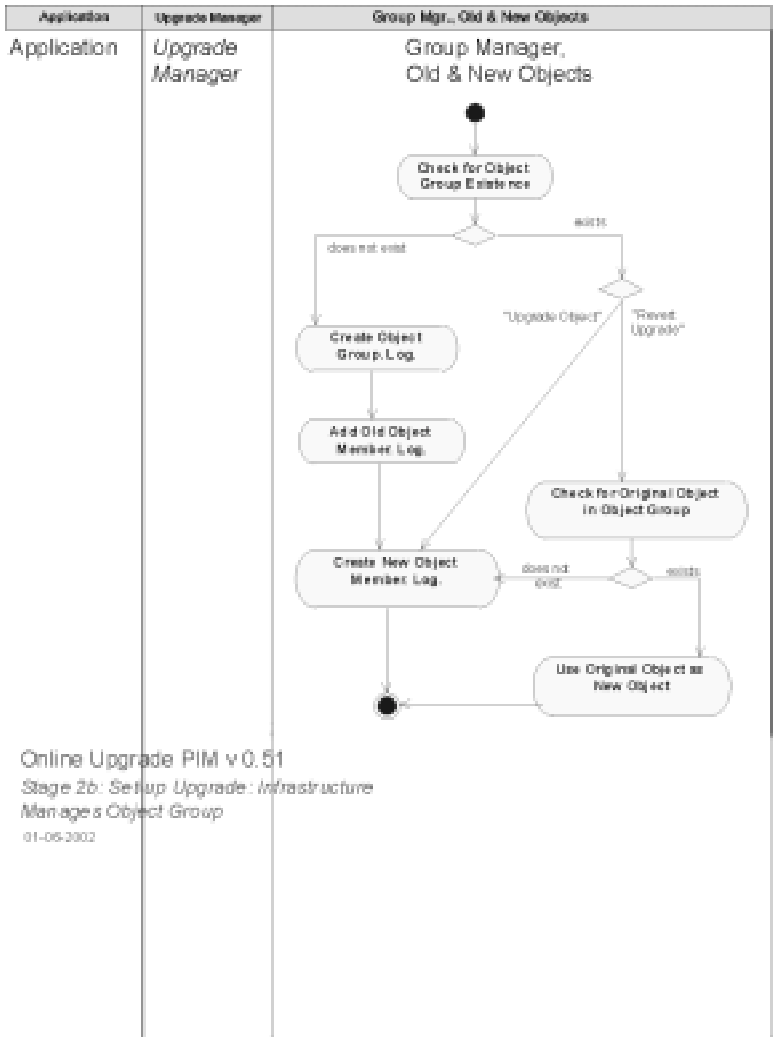
Class Diagram
01-06-2002

Online Upgrade PIM v 0.51

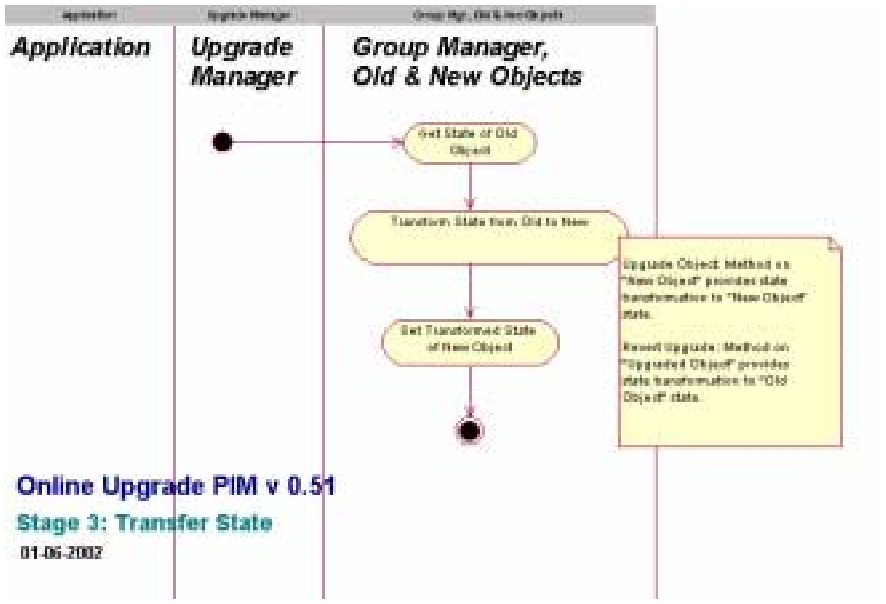

Application
Upgrade Manager
Group Manager, Old & New Objects
0. Upgrade Manager Online
1. Old Upgradeable Object Online
Invoke "Upgrade Object" with Upgrade Style Specified
2. Set-up Upgrade
Fail
Succeed
3. Transfer State
Fail
Succeed
4. Issue Commit
Fail
Succeed
Rollback
5. Cleanup & Free Recovery Resources
Evaluate Upgrade
Accept
Reject
Invoke "Revert Upgrade"
Upgrade Styles:
A. Infrastructure Controlled with Implicit Commit and Implicit Rollback
B. Infrastructure Controlled with Explicit Commit and Explicit Rollback
C. Application Controlled with Explicit Commit and Explicit Rollback
Revert Upgrade Style:
A. Infrastructure Controlled with Implicit Commit and Implicit Rollback
Online Upgrade PIM v 0.51
Top Level
01-06-2002

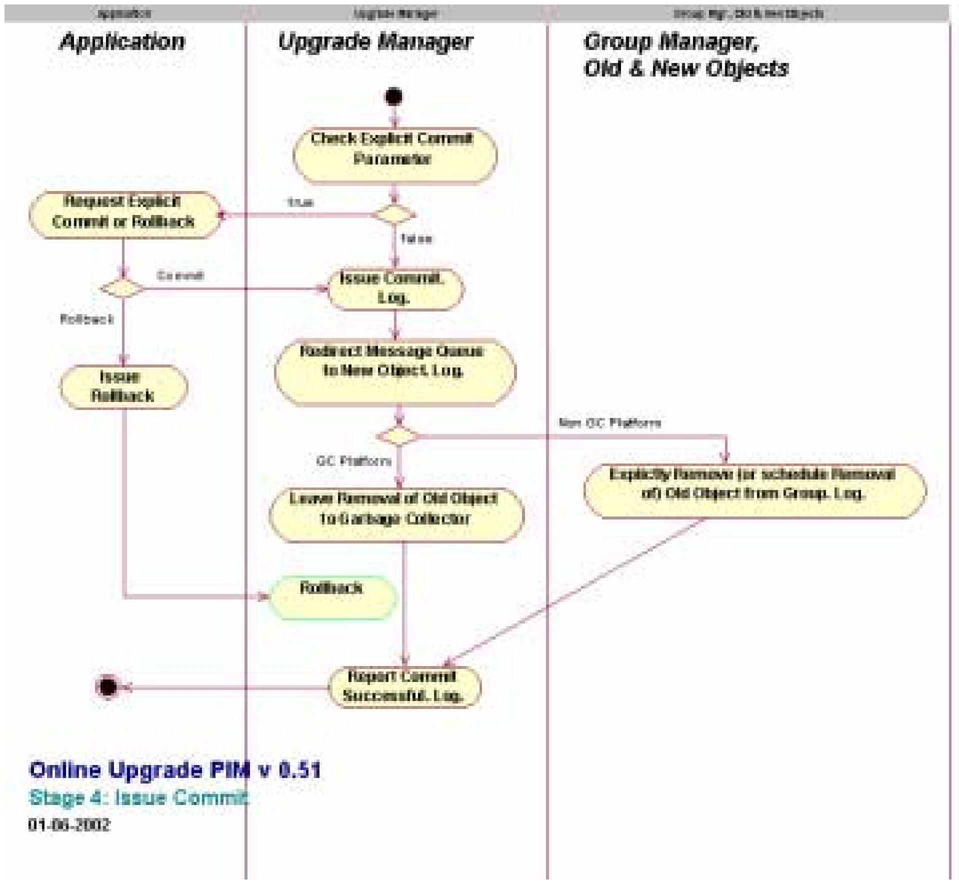

Application
Upgrade Manager
Group Manager, Old & New Objects
Check Log for Incomplete Upgrade (requires Rollback)
exists
does not exist
Execute Rollback Recovery
Accept New Requests
Online Upgrade PIM v 0.51
Stage 0.
Upgrade Manager Online
01.06.2002

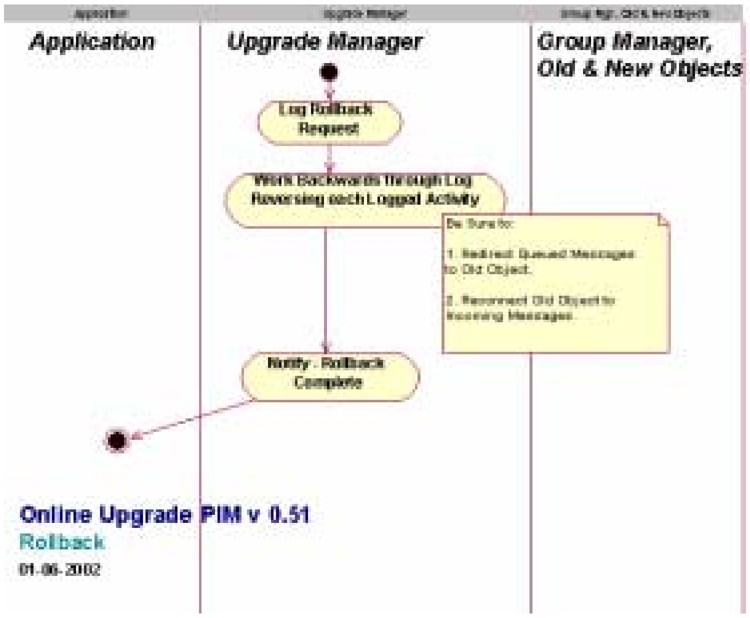

Application (from 3. Transfer State)
Upgrade Manager (from 3. Transfer State)
Group Manager
Application
Upgrade Manager
Group Manager,
Old & New Objects
Log
Check Revert from Object for New-to-Old Transform State Method
Check Old Object - Upgradeable ?
Upgrade Controlled by ?
Application Manages Object Group
Object Group must conta...
Infrastructure
Infrastructure Manages Object Group
Application
Check Interface Compatibility between Old and New Objects
Notify Old Object to Get Ready to Pause
I_am_ready()
I_am_also_ready()
Set-up Message Queue Recovery Resources. Log.
e.g. Replicate or Persist Message Queue
Intercept & Queue Messages for Old Object. Log.
Pause
Old Object Paused. Log.
Online Upgrade PIM v 0.51
Stage 2: Set-up Upgrade



Application
Upgrade Manager
Group Mgr., Old & New Objects
Application
Upgrade Manager
Group Manager, Old & New Objects
Check for Object Group Existence
exists
does not exist
"Upgrade Object"
"Revert Upgrade"
Create Object Group Log
Add Old Object Member Log
Check for Original Object in Object Group
Create New Object Member Log
does not exist
exists
Use Original Object as New Object
Online Upgrade PIM v 0.51
Stage 2b: Setup Upgrade: Infrastructure
Manages Object Group
01-06-2002


Application
Upgrade Manager
Group Manager, Old & New Objects
Get State of Old Object
Transform State from Old to New
Set Transformed State of New Object
Upgrade Object: Method on "New Object" provides state transformation to "New Object" state.
Revert Upgrade: Method on "Upgraded Object" provides state transformation to "Old Object" state.
Online Upgrade PIM v 0.51
Stage 3: Transfer State
01-06-2002


Application
Upgrade Manager
Upgrade Manager
Group Manager,
Old & New Objects
Check Explicit Commit Parameter
true
false
Request Explicit Commit or Rollback
Commit
Rollback
Issue Commit. Log.
Redirect Message Queue to New Object. Log.
Issue Rollback
Non GC Platforms
GC Platforms
Leave Removal of Old Object to Garbage Collector
Explicitly Remove (or schedule Removal of) Old Object from Group. Log.
Rollback
Report Commit Successful. Log.
Online Upgrade PIM v 0.51
Stage 4: Issue Commit
01-06-2002


Application
Upgrade Manager
Group Manager, Old & New Objects
Log Rollback Request
Work Backwards through Log Reversing each Logged Activity
Be Sure to:
1. Redirect Queued Messages to Old Object.
2. Reconnect Old Object to Incoming Messages.
Notify - Rollback Complete
Online Upgrade PIM v 0.51
Rollback
01-06-2002

Group Management 3

3.1 Overview

This chapter defines the **GroupManager** interface. The Group Manager is responsible for managing object groups. For Online Upgrades, object groups are heterogeneous in that the members of the group are an instance of the old implementation of the object and the corresponding instance of the new implementation of the object. For Fault Tolerant CORBA, object groups are homogeneous in that the members of the group are the replicas of an object. If online upgrades are combined with fault tolerance, an object group may contains replicas of the instance of the old implemenation and replicas of the instance of the new implementation and, thus, the object groups are heterogeneous.

The **GroupManager** interface extends the **PortableGroup** module, which is also defined in this chapter. The **PortableGroup** module contains three interfaces: **PropertyManager**, **GenericFactory** and **ObjectGroupManager**. The intention of the **PortableGroup** module is provide a single module with group management methods that might be used by Fault Tolerant CORBA, Unreliable Multicast, Data Parallel CORBA, Load Balancing, Online Upgrades and perhaps other future OMG specifications.

The **PropertyManager** interface allows properties of the object groups to be set, namely, the MembershipStyle and the Factories. The properties may be set statically as defaults for the particular application or for a particular type, or may be set or changed dynamically while the application is executing.

The **GenericFactory** interface is used by the application to create object groups. It is also used by the Group Manager to create individual members of an object group.

For the infrastructure-controlled MembershipStyle, the application uses the **GenericFactory** interface of the Group Manager to create an object group. The Group Manager, in turn, invokes the individual factories, for the appropriate locations, to create the members of the object group. The Group Manager adds the members to the object group and creates the object group reference. Subsequently, the Group Manager removes members, if necessary.

For the application-controlled MembershipStyle, the application uses the **ObjectGroupManager** interface of the Group Manager to create a member of an object group, to add an existing object to an object group, or to remove a member from an object group, citing the location of the member to be created, added or removed. It also allows the application to query the locations of the members of an object group.

3.2 PortableGroup Module

3.2.1 Properties of Portable Group

Each object group has an associated set of properties that are set as defaults for the particular application, that are set for the type of the object, that are set when the object group is created, or that are set subsequently while the application executes. The names and values of the specified properties are given below..

3.2.1.1 MembershipStyle

Name	org.omg.pg.MembershipStyle
Value	PG::MEMB_APP_CTRL
	PG::MEMB_INF_CTRL

If the value of the MembershipStyle is MEMB_APP_CTRL, the application may create an object itself and then invoke the **add_member()** method of the **ObjectGroupManager** interface to cause the Group Manager to add the object to the object group. Alternatively, the application may invoke the **create_member()** method of the **ObjectGroupManager** interface to cause the Group Manager to create the member and add it to the object group.

If the value of the MembershipStyle is MEMB_INF_CTRL, the Group Manager invokes the individual factories, for the appropriate locations, to create the members of the object group.

3.2.1.2 Factories

Name	org.omg.pg.Factories
Value	FactoryInfos

A factory is an object, the purpose of which is to create other objects. FactoryInfos is a sequence of FactoryInfo, where FactoryInfo contains the reference to the factory, the location at which the factory is to create a member of the object group and criteria that the factory is to use to create the member.

3.2.2 Common Types

```
module PortableGroup {

    // Specification of Interoperable Object Group Reference
    typedef string DomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;
    struct TagGroupTaggedComponent { // tag = TAG_GROUP
        GIOP::Version version;
        DomainId domain_id;
        ObjectGroupRefVersion object_group_ref_version;
    };

    // Specification of Common Types and Exceptions for Group Management
    interface GenericFactory;

    typedef CORBA::RepositoryId TypeId;
    typedef Object ObjectGroup;

    typedef string Name;
    typedef any Value;
    struct Property {
            Name nam;
            Value val;
    };
    typedef sequence<Property> Properties;

    typedef CosNaming::Name Location;
    typedef sequence<Location> Locations;
    typedef Properties Criteria;
    struct FactoryInfo {
          GenericFactory the_factory;
          Location the_location;
          Criteria the_criteria;
    };
    typedef sequence<FactoryInfo>  FactoryInfos;

    typedef unsigned short MembershipStyleValue;
    const MembershipStyleValue MEMB_APP_CTRL = 0;
    const MembershipStyleValue MEMB_INF_CTRL = 1;

    typedef FactoryInfos FactoriesValue;

    exception InterfaceNotFound {};
    exception ObjectGroupNotFound {};
    exception MemberNotFound {};
    exception ObjectNotFound {};
    exception MemberAlreadyPresent {};
    exception ObjectNotCreated {};
```

```
        exception ObjectNotAdded {};
        exception UnsupportedProperty {
                Name nam;
                Value val;
        };
        exception InvalidPropertyValue {
                Name nam;
                Value val;
        };
        exception NoFactory {
                Location the_location;
                TypeId type_id;
        };
        exception InvalidCriteria {
                Criteria invalid_criteria;
        };
        exception CannotMeetCriteria {
                Criteria unmet_criteria;
        };
};
```

3.2.2.1 Identifiers

typedef Object ObjectGroup;

A reference to an object group.

typedef string Name;

The name of a property.

typedef any Value;

The value of a property.

```
struct Property {
     Name nam;
     Value val;
};
```

The name-value pair for a property.

typedef sequence<Property> Properties;

A sequence of properties.

typedef CosNaming::Name Location;

The name for a host, device, cluster of hosts, etc., which may be hierarchical. For example, for a variable loc of the type location, loc[0].kind might be “HostIP”, loc[0].id might be an IP address, loc[1].kind might be “ProcessID” and loc[1].id might be a process identifier.

typedef sequence<Location> Locations;

A sequence of locations of the members of an object group.

typedef Properties Criteria;

Criteria is a sequence of property, i.e., name-value pair. Examples of criteria are initialization values, constraints on an object, preferred location of the object.

```
struct FactoryInfo {
        GenericFactory factory;
        Location the_location;
        Criteria the_criteria;
    };
```

A structure that contains the reference to a factory, the location at which the factory is to create an object and the criteria, such as initialization values, constraints on the object, etc., which the factory is to use to create the object.

typedef sequence<FactoryInfo> FactoryInfos;

A sequence of FactoryInfos.

```
typedef unsigned short MembershipStyleValue;
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;
```

The values of the MembershipStyle property.

typedef FactoryInfos FactoriesValue;

The value of the Factories property.

3.2.2.2 Exceptions

exception InterfaceNotFound {};

The object with the given interface is not found.

exception ObjectGroupNotFound {};

The object group with the given identifier is not found.

exception MemberNotFound {};

No member of the object group exists at the given location.

exception ObjectNotFound {};

The object is not found.

exception MemberAlreadyPresent {};

A member of the object group already exists at the given location.

exception ObjectNotCreated {};

The object was not created.

exception ObjectNotAdded {};

The object was not added to the object group.

```
exception UnsupportedProperty {
        Name nam;
};
```

A property named in the property sequence is not supported.

```
exception InvalidPropertyValue {
        Name nam;
        Value val;
};
```

A value of a property in the property sequence is not valid either in itself (for example, because the initial number of members is negative) or because it conflicts with another value of a property in the sequence or with other property values already in effect that are not overridden.

```
exception NoFactory {
          Location the_location;
          TypeId type_id;
};
```

The factory that is to create an object at the given location with the given repository identifier does not exist.

```
exception InvalidCriteria {
           Criteria invalid_criteria;
};
```

The factory does not understand the given criteria.

```
exception CannotMeetCriteria {
        Criteria unmet_criteria;
};
```

The factory understands the given criteria, but cannot satisfy the criteria.

3.2.3 PropertyManager

The **PropertyManager** interface of the **PortableGroup** module provides methods that allow the user to set properties of object groups, which for online upgrades are the MembershipStyle and the Factories. It may set the properties statically as defaults for a particular application or for a particular type, or may set or change the properties dynamically while the application is executing.

```
module PortableGroup {

    // Specification of the Property Manager interface
    interface PropertyManager {

        void set_default_properties
            (in Properties props)
        raises
            (InvalidPropertyValue,
             UnsupportedProperty);

        Properties get_default_properties();

        void remove_default_properties
            (in Properties props)
        raises
            (InvalidPropertyValue,
             UnsupportedProperty);

        void set_type_properties
            (in TypeId type_id,
             in Properties overrides)
        raises
            (InvalidPropertyValue,
             UnsupportedProperty);

        Properties get_type_properties
            (in TypeId type_id);

        void remove_type_properties
            (in TypeId type_id,
             in Properties props)
        raises
            (InvalidPropertyValue,
             UnsupportedProperty);

        void set_properties_dynamically
            (in ObjectGroup object_group,
             in Properties overrides)
        raises
            (ObjectGroupNotFound,
             InvalidPropertyValue,
             UnsupportedProperty);

        Properties get_properties
            (in ObjectGroup object_group)
        raises
            (ObjectGroupNotFound);
    };
};
```

3.2.3.1 Operations

set_default_properties

The method sets the default properties for all object groups that are to be created within the application.

```
void set_default_properties
      (in Properties props)
raises
      (InvalidPropertyValue,
       UnsupportedProperty);
```

Parameters

props	The properties to be set for all newly created object groups within the application.

Raises

InvalidPropertyValue if one or more of the values of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

get_default_properties

The method returns the default properties for the object groups within the particular application.

```
Properties get_default_properties();
```

Return Value

The default properties that have been set for the object groups.

remove_default_properties

The method removes the given default properties.

```
void remove_default_properties
      (in Properties props)
raises
      (InvalidPropertyValue,
       UnsupportedProperty);
```

Parameters

props	The properties to be removed.

Raises

InvalidPropertyValue if one or more of the values of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

set_type_properties

The method sets the properties that override the default properties of the object groups, with the given type identifier, that are created in the future.

```
void set_type_properties
      (in TypeId type_id,
       in Properties overrides)
raises
      (InvalidPropertyValue,
       UnsupportedProperty);
```

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set.
overrides	The overriding properties.

Raises

InvalidPropertyValue if one or more of the values of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

get_type_properties

The method returns the properties of the object groups, with the given type identifier, that are created in the future. These properties include the properties determined by **set_type_properties()**, as well as the default properties that are not overridden by **set_type_properties()**.

```
Properties get_type_properties
      (in TypeId type_id);
```

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set.

Return Value

The effective properties for the given type identifier.

remove_type_properties

The method removes the given properties, with the given type identifier.

```
void remove_type_properties
    (in TypeId type_id,
     in Properties props)
raises
    (InvalidPropertyValue,
     UnsupportedProperty);
```

Parameters

type_id	The repository id for which the given properties are to be removed.
props	The properties to be removed.

Raises

InvalidPropertyValue if one or more of the values of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

set_properties_dynamically

The method sets the properties for the object group with the given reference dynamically while the application executes. The properties given as a parameter override the properties for the object when it was created which, in turn, override the properties for the given type which, in turn, override the default properties.

```
void set_properties_dynamically
    (in ObjectGroup object_group,
     in Properties overrides)
raises
    (ObjectGroupNotFound,
      InvalidPropertyValue,
      UnsupportedProperty);
```

Parameters

object_group	The reference of the object group for which the overriding properties are set.
overrides	The overriding properties.

Raises

InvalidPropertyValue if one or more of the values of the properties in the sequence is not valid.

UnsupportedProperty if one or more of the properties in the sequence is not supported.

get_properties

The method returns the current properties of the given object group. These properties include those that are set dynamically, those that are set when the object group was created but are not overridden by **set_properties_dynamically()**, those that are set as properties of a type but are not overridden by **create_object()** and **set_properties_dynamically()**, and those that are set as defaults but are not overridden by **set_type_properties()**, **create_object()** and **set_properties_dynamically()**.

```
Properties get_properties
    (in ObjectGroup object_group)
raises
    (ObjectGroupNotFound)
```

Parameters

object_group	The reference of the object group for which the properties are to be returned.

Return Value

The set of current properties for the object group with the given reference.

Raises

ObjectGroupNotFound if the object group is not found.

3.2.4 ObjectGroupManager

The **ObjectGroupManager** interface provides methods that allow an application to exercise control over the addition, removal and locations of members of an object group and to obtain the current reference and identifier for an object group.

```
module PortableGroup {

    // Specification of the Object Group Manager interface
    interface ObjectGroupManager {

        ObjectGroup create_member
            (in ObjectGroup object_group,
             in Location the_location,
             in TypeId type_id,
             in Criteria the_criteria)
        raises
            (ObjectGroupNotFound,
             MemberAlreadyPresent,
             NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             CannotMeetCriteria);

        ObjectGroup add_member
            (in ObjectGroup object_group,
             in Location the_location,
             in Object member)
        raises
            (ObjectGroupNotFound,
             MemberAlreadyPresent,
             ObjectNotAdded);

        ObjectGroup remove_member
            (in ObjectGroup object_group,
             in Location the_location)
        raises
            (ObjectGroupNotFound,
             MemberNotFound);

        Locations locations_of_members
            (in ObjectGroup object_group)
        raises
            (ObjectGroupNotFound);

        ObjectGroupId get_object_group_id
            (in ObjectGroup object_group)
        raises
            (ObjectGroupNotFound);
        ObjectGroup get_object_group_ref
            (in ObjectGroup object_group)
        raises(ObjectGroupNotFound);

        Object get_member_ref
            (in ObjectGroup object_group,
             in Location the_location)
        raises
            (ObjectGroupNotFound,
             MemberNotFound);
    };
};
```

3.2.4.1 Operations

create_member

The **create_member()** method allows the application to exercise explicit control over the creation of a member of an object group, and to determine where the member is created.

```
        ObjectGroup create_member
            (in ObjectGroup object_group,
             in Location the_location,
             in TypeId type_id,
             in Criteria the_criteria)
        raises
            (ObjectGroupNotFound,
             MemberAlreadyPresent,
             NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             CannotMeetCriteria);
```

Parameters

object_group	The object group reference for the object group to which the member is to be added.
the_location	The physical location, i.e., a host, cluster of hosts, etc., at which the new member is to be created. There is at most one member of an object group at each location.
type_id	The repository identifier for the type of the object.
the_criteria	Parameters to be passed to the factory, which the factory evaluates before creating the object. The criteria are implementation-specific and are not defined in this specification. Examples of criteria are initialization values, constraints on the member, etc. The criteria passed in as a parameter to **create_member()**, if any, override the criteria set in the FactoryInfos property of the given object group for the given location.

Return Value

The object group reference of the object group with the member added. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found.

MemberAlreadyPresent if a member of the object group already exists at the given location.

NoFactory if a factory that is capable of creating a member of the object group with the given type_id and at the given location cannot be found.

ObjectNotCreated if the factory cannot create the member and add it to the object group.

InvalidCriteria if the factory does not understand the criteria

CannotMeetCriteria if the factory understands the criteria but cannot satisfy it.

add_member

The **add_member()** method allows an application to exercise explicit control over the addition of an existing object to an object group at a particular location.

```
ObjectGroup add_member
    (in ObjectGroup object_group,
     in Location the_location,
     in Object member)
raises
    (ObjectGroupNotFound,
     MemberAlreadyPresent,
     ObjectNotAdded);
```

Parameters

object_group	The object group reference of the object group to which the existing object is to be added.
the_location	The physical location, i.e., a host, cluster of hosts, etc., of the object to be added. There is at most one member of an object group at each location.
member	The reference of the object to be added.

Return Value

The object group reference for the object group with the object added. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found.

MemberAlreadyPresent if a member of the object group already exists at the given location.

ObjectNotAdded if the object is not added to the object group.

remove_member

The **remove_member()** method allows an application to exercise explicit control over the removal of a member from an object group at a particular location.

If the application invoked the **create_object()** method of the **GenericFactory** interface to create the member object and used the **add_member()** method to add the object to the object group, when the application invokes **remove_member()**, the member is removed from the group but is not deleted. Deletion of the object is the responsibility of the application.

If the application invoked the **create_member()** method to create the member object, when the application invokes the **remove_member()** method to remove the member from the object group, the member is first removed from the object group and then the **delete_object()** method of the **GenericFactory** interface is invoked to delete the object.

If the member was created by the **create_object()** method of the **GenericFactory** interface, when the application invokes the **remove_member()** method to remove the member, the member is first removed from the group and then the **delete_object()** method of the **GenericFactory** interface is invoked to delete the object.

```
ObjectGroup remove_member
      (in ObjectGroup object_group,
       in Location the_location)
raises
      (ObjectGroupNotFound,
       MemberNotFound};
```

Parameters

object_group	The object group reference for the object group from which the member is to be removed.
the_location	The physical location, i.e., a host, cluster of hosts, etc., of the member to be removed.

Return Value

The object group reference for the object group with the member removed. This reference may be the same as that passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found.

MemberNotFound if the member of the object group is not found at the given location.

locations_of_members

The **locations_of_members()** method allows the application to determine the locations of the members of the given object group.

```
Locations locations_of_members
    (in ObjectGroup object_group)
raises
    (ObjectGroupNotFound);
```

Parameters

object_group	The object group reference of the object group.

Return Value

A sequence of locations at which the members of the object group currently exist.

Raises

ObjectGroupNotFound if the object group is not found.

get_object_group_id

The **get_object_group_id()** method takes a reference for an object group as an in parameter, and returns the identifier of the object group.

```
ObjectGroupId get_object_group_id
    (in ObjectGroup object_group)
raises
    (ObjectGroupNotFound);
```

Parameters

object_group	An object group reference for the object group.

Return Value

The object group identifier for the object group.

Raises

ObjectGroupNotFound if the object group is not found.

get_object_group_ref

The **get_object_group_ref()** method takes a reference for an object group as an in parameter, and returns the current reference for the object group.

```
ObjectGroup get_object_group_ref
    (in ObjectGroup object_group)
raises
    (ObjectGroupNotFound);
```

Parameters

object_group	An object group reference for the object group.

Return Value

The current object group reference for the object group. The returned reference may be the same as the reference passed in as a parameter.

Raises

ObjectGroupNotFound if the object group is not found.

get_member_ref

The **get_member_ref()** method takes a reference for an object group and a location as in parameters, and returns a reference for the member.

```
Object get_member_ref
      (in ObjectGroup object_group,
       in Location the_location)
raises
      (ObjectGroupNotFound,
       MemberNotFound);
```

Parameters

object_group	An object group reference for an object group.
the_location	The location of the member.

Return Value

The reference for the member.

Raises

ObjectGroupNotFound if the object group is not found.

MemberNotFound if the member is not found.

3.2.5 GenericFactory

The **GenericFactory** interface is generic in that it allows the creation of object groups, members of object groups, and individual objects (that are not members of object groups).

```
module PortableGroup {

    // Specification of the GenericFactory interface
    interface GenericFactory {
        typedef any FactoryCreationId;

        Object create_object
            (in TypeId type_id,
             in Criteria the_criteria,
             out FactoryCreationId factory_creation_id)
        raises
            (NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             InvalidPropertyValue,
             CannotMeetCriteria);

        void delete_object
            (in FactoryCreationId factory_creation_id)
        raises
             (ObjectNotFound);
    };
};
```

The **create_object()** method takes a type_id as an in parameter. It also takes the_criteria as an in parameter, which allows a user to specify additional criteria, such as initialization values for the object implementation, constraints on the object, or preferred location of the object. The type_id and the_criteria in parameters of the **create_object()** method contribute to the genericity and the flexibility of the **GenericFactory** interface.

The **create_object()** method has an out parameter, factory_creation_id, that is retained by the entity that invoked the method so that it can later invoke the **delete_object()** method of the factory using the factory_creation_id as an in parameter, to cause the factory to delete the object. The factory must also retain this identification information so that it can actually delete the object.

Each implementation of the **GenericFactory** interface may create objects of one or more types at one or more locations.

3.2.5.1 Identifiers

typedef any FactoryCreationId;

An identifer that is assigned to an object by the factory that creates the object and that is used by the factory to delete the object subsequently.

3.2.5.2 Operations

create_object

The method creates an object, using the type_id parameter to determine which type of object to create and the_criteria parameter to determine restrictions on how and where to create the object. The out parameter, factory_creation_id, allows the entity that invoked the factory, and the factory itself, to identify the object for subsequent deletion.

```
Object create_object
      (in TypeId type_id,
       in Criteria the_criteria,
       out FactoryCreationId factory_creation_id)
raises
      (NoFactory,
       ObjectNotCreated,
       InvalidCriteria,
       InvalidPropertyValue,
       CannotMeetCriteria);
```

Parameters

type_id	The repository identifier of the object to be created by the factory.
the_criteria	Information passed to the factory, which the factory evaluates before creating the object. Examples of criteria are initialization values, constraints on the object, preferred location of the object, etc.
factory_creation_id	An identifier that allows the factory to delete the object subsequently.

Return Value

The reference to the object created by the GenericFactory.

Raises

NoFactory if the object cannot be created.

ObjectNotCreated if the factory cannot create the object

InvalidCriteria if the factory does not understand the criteria

InvalidPropertyValue if the value of a property passed in as criteria is not valid

CannotMeetCriteria if the factory understands the criteria but cannot satisfy it.

delete_object

The method deletes the object with the given identifier.

```
void delete_object
    (in FactoryCreationId factory_creation_id)
raises
    (ObjectNotFound);
```

Parameters

factory_creation_id	An identifier for the object that is to be deleted.

Raises

ObjectNotFound if the object cannot be found.

3.3 *Properties of Upgradeable Objects*

Each object group for an Upgradeable object has three properties beyond the properties defined by the **PortableGroup** module, namely, Quiescence, PauseInterval and RemovalInterval.

3.3.1 *Quiescence*

Name	org.omg.ou.Quiescence
Value	boolean QuiescenceStyleValue

If the value of the QuiescenceStyle property is false, the Upgradeable object uses no callbacks and the **are_you_ready()** method operates as described below, with queuing of request messages when **are_you_ready()** is invoked. If the Upgradeable object uses callbacks, this strategy could result in a deadlock. Thus, if the value of the QuiescenceStyle property is true, the Upgradeable object uses callbacks and the queuing of request messages starts only when the object invokes the **i_am_ready()** method with the **ready** parameter set to true. It is the responsibility of the Upgrade Mechanisms to ensure that no request messages are delivered to the object after the object has invoked **i_am_ready()** with the **ready** parameter equal to true.

3.3.2 *PauseInterval*

Name	org.omg.ou.PauseInterval
Value	long

If the object being upgraded invokes **i_am_ready()** with the value of **ready** equal to false, the Upgrade Manager may wait for an interval and then reinvoke **are_you_ready()**. The PauseInterval defines the length of the interval, in seconds, that the Upgrade Manager waits before invoking **are_you_ready()** again. If the object being upgraded invokes **i_am_ready()** with the **ready** parameter equal to false and the PauseInterval is 0, the Upgrade Manager does not reinvoke **are_you_ready()**, but immediately rolls back the upgrade.

3.3.3 RemovalInterval

Name	org.omg.ou.RemovalInterval
Value	long

When an object is upgraded successfully, the instance of the old implementation of the object is removed by the Upgrade Manager. The user, having observed unsatisfactory behavior from the instance of the new implementation of the object, may decide to invoke **revert_upgrade()** in order to restore the instance of the old implementation of the object. Consequently, the Upgrade Manager may delay the removal of the old implementation for a short period so that, if the user does invoke **revert_upgrade()**, the reversion can be performed more quickly. The RemovalInterval defines the length of the interval, in seconds, for which the instance of the old implementation is retained before it is removed.

3.4 GroupManager Interface

The **GroupManager** interface, defined below, is one of the interfaces of the **OnlineUpgrades** module. The other two interfaces, the **UpgradeManager** interface and the **Upgradeable** interface, are defined in subsequent chapters.

The **GroupManager** interface extends the **PortableGroup** module and inherits the methods of the **PropertyManager**, **ObjectGroupManager** and **GenericFactory** interfaces. The methods of the **PropertyManager** interface allow definition of properties associated with object groups that the Group Manager creates. The methods of the **ObjectGroupManager** interface allow an application to exercise control over the addition, removal and location of members of an object group. The methods of the **GenericFactory** interface allow the Group Manager to create and delete object groups.

```
module  OnlineUpgrades {

    // Specification of Types for Upgrade Management Properties
    typedef boolean QuiescenceStyleValue;
    typedef long PauseIntervalValue;
    typedef long RemovalIntervalValue;

    // Specification of the GroupManager interface
    interface GroupManager :
        PortableGroup::PropertyManager,
        PortableGroup::ObjectGroupManager,
        PortableGroup::GenericFactory {

            void pause_member
                (in ObjectGroup object_group,
                 in Location the_location)
            raises
                (MemberNotFound);

            ObjectGroup resume_member
                (in ObjectGroup object_group,
                 in Location the_location)
            raises
                (MemberNotFound);
    };
};
```

3.4.1 Identifiers

typedef boolean QuiescenceStyleValue;

If the value of the QuiescenceStyle property is false, the Upgradeable object uses no callbacks and the **are_you_ready()** method operates with queuing of request messages when **are_you_ready()** is invoked. If the value of the QuiescenceStyle property is true, the Upgradeable object uses callbacks and the queuing of request messages starts only when the object invokes the **i_am_ready()** method with the **ready** parameter set to true.

typedef long PauseIntervalValue;

The value of the PauseInterval property. The value is the length of the interval, in seconds, between successive invocations of **are_you_ready()** by the Upgrade Manager, if the object invokes **i_am_ready()** with the **ready** parameter equal to false.

typedef long RemovalIntervalValue;

The value of the RemovalInterval property. The value is the length of the interval, in seconds, for which the instance of the old implementation is retained before the Upgrade Manager removes it.

3.4.2 *Operations*

3.4.2.1 *pause_member*

The method takes the object group reference for the object being upgraded, and the location of the instance of the old implementation, as in parameters. It is invoked by the Upgrade Manager on the Group Manager, when the instance of the old implementation invokes **i_am_ready()** with the **ready** parameter equal to true.

```
void pause_member
      (in ObjectGroup object_group,
       in Location the_location)
raises
      (MemberNotFound);
```

Parameters

object_group	An object group reference for the object being upgraded.
the_location	The location of the member of the object group that corresponds to the instance of the old implementation.

Raises

MemberNotFound if the member is not found.

3.4.2.2 *resume_member*

The method takes an object group reference for the object being upgraded, and the location of the instance of the new upgraded implementation, as in parameters, and returns an object group reference containing a profile for the instance of the new implementation. It is invoked by the Upgrade Manager on the Group Manager to start the instance of the new implementation processing requests.

```
ObjectGroup resume_member
      (in ObjectGroup object_group,
       in Location the_location)
raises
      (MemberNotFound);
```

Parameters

object_group	An object group reference for the object being upgraded.
the_location	The location of the member of the object group that corresponds to the instance of the new upgraded implementation.

Return Value

An object group reference that contains a profile for the instance of the new upgraded implementation.

Raises

MemberNotFound if the member is not found.

3.4.3 Usage

The **GroupManager** interface inherits the **GenericFactory** interface to allow the application to invoke the Group Manager to create object groups. The application's local factory objects implement the **GenericFactory** interface to allow the Group Manager to invoke the methods of that interface to create individual members of an object group and also to create individual objects. The Group Manager is programmed by the vendor of the Upgrade infrastructure, and the application's local factory objects are programmed by the application programmer.

If the MembershipStyle is MEMB_INF_CTRL, the Upgrade Manager invokes the **create_object()** method of the **GenericFactory** interface of the Group Manager, which creates an object group and returns an object group reference.

If the MembershipStyle is MEMB_APP_CTRL, the application invokes the **create_object()** method of the **GenericFactory** interface of the Group Manager, which creates an object group with no members and returns an object group reference. In this case, the object group reference contains a TAG_MULTIPLE_COMPONENTS profile with a TAG_PG_GROUP component in it, rather than TAG_INTERNET_IOP profiles.

If the Membership Style is MEMB_INF_CTRL, the Upgrade Manager invokes the **create_member()** method of the Group Manager which, in turn, invokes the **create_object()** method of the **GenericFactory** interface of the application's local factories to create a member of the object group and then adds the member to the group, as shown in Figure 3-1. It uses the locations to choose a factory from the Factories sequence and uses the factory reference to invoke the method.

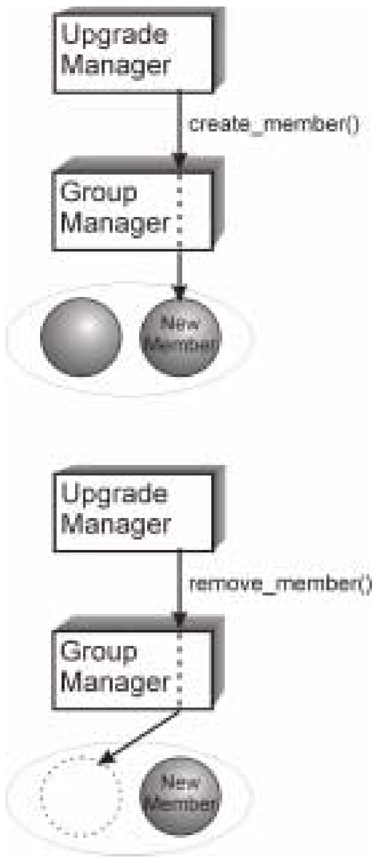


Figure 3-1 Infrastructure-controlled Membership Style

If the Membership Style is MEMB_APP_CTRL, the application may invoke the **create_member()** method of the Group Manager which, in turn, invokes the **create_object()** method of the **GenericFactory** interface of the application's local factory and then adds the member to the group. Alternatively, the application may invoke the **create_object()** method of the **GenericFactory** interface of the application's local factory to create the object and may then invoke the **add_member()** method of the Group Manager to add the object to the group, as shown in Figure 3-2.

The **create_object()** method of the application's local factory accepts a criterion with the name org.omg.pg.ObjectLocation (which is reserved for specifying the location at which the factory is to create the object). The value of this criterion instructs the factory where to create the object. The **create_object()** method of the Group Manager accepts properties within the_criteria parameter. These properties are contained in a single criterion with the reserved name org.omg.pg.PGProperties. Such properties, if any, override the corresponding properties that are specified as defaults or based on the type of the object. The Group Manager removes the org.omg.pg.PGProperties criterion from the_criteria parameter, adds the org.omg.pg.ObjectLocation criterion, and appends any location-specific criteria (specified in the Factories property for the particular location) to the_criteria parameter before it invokes **create_object()** on the application's local factory.

The **create_object()** method may raise the NoFactory exception. For the application's local factory object, the raised exception indicates that the factory cannot create an individual object of the type_id at the location. For the Group Manager, the raised exception indicates that the Group Manager cannot create the object group because it cannot find a factory that is capable of constructing a member of the object group of the type_id at the location.

The **delete_object()** method may be invoked by the application or the Group Manager. on the application's local factory. In this case, the application's local factory deletes a single object.

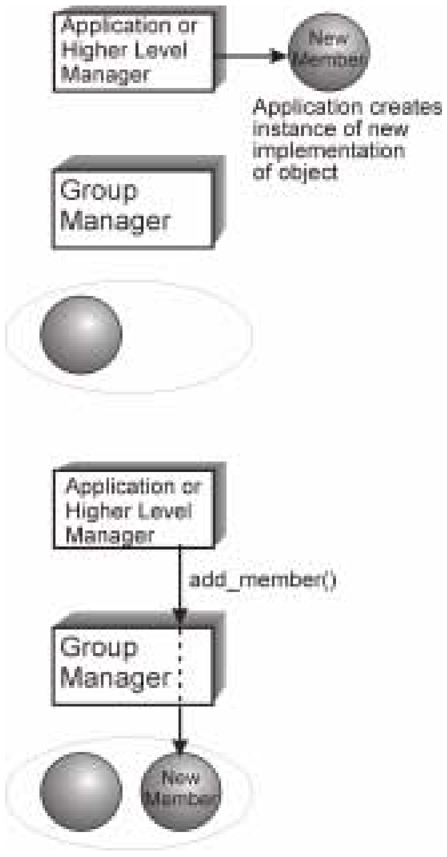

Application or Higher Level Manager
New Member
Application creates instance of new implementation of object
Group Manager
Application or Higher Level Manager
add_member()
Group Manager
New Member

Figure 3-2 Application-controlled Membership Style

Upgrade Management 4

4.1 Overview

The **UpgradeManager** interface, defined in this chapter, is the principal management interface for online upgrades of objects. It provides methods to prepare an object for upgrading, to perform the upgrades of one or more objects, to rollback upgrades of objects, and to revert an object from its new implementation to its old implementation.

The Upgrade Manager object, which implements this interface, uses the methods of the **GroupManager** interface to create new instances of, and to manipulate, the object that is being upgraded. The Group Manager object, which implements the interfaces of the **PortableGroup** module, can also be used to exercise more precise application control over the instantiation of new implementations of the object that is being upgraded.

The Upgrade Manager also invokes the **get_state()** method of the **Upgradeable::Checkpointable** interface and the **are_you_ready()** and **transform_and_set_state()** methods of the **Upgradeable** interface, which the object being upgraded must implement. These methods allow the Upgrade Manager to establish that the object is in a safe and quiescent state before being upgraded, and to transfer the state of an instance of the old implementation of the object to an instance of the new implementation of the object.

4.2 UpgradeManager Interface

The **UpgradeManager** interface provides five methods: **upgrade_object()**, **commit_upgrade()**, **rollback_upgrade()**, **revert_upgrade()** and **i_am_ready()**.

The most important of these methods is the **upgrade_object()** method, which initiates the upgrading of an object. The parameters of the method allow the user to define the object-to-be-upgraded, its type, the location at which to create the instance of the new implementation, and criteria to be used in its creation, including the factory to be used. The Upgrade Manager creates the instance of the new implementation and pauses the instance of the old implementation. The Upgrade Mechanisms retrieve the state from the instance of the new implementation, transform in into the state of the instance of the new implementation, and assign it to the instance of the new implementation. Another parameter determines whether the upgrade is committed automatically or whether it must wait until the user invokes the **commit_upgrade()** method of the **UpgradeManager** interface.

The **commit_upgrade()** method allows the user to prepare several objects for upgrading and then to upgrade all of them together. The method commits all of the objects for which it has been invoked but for which the commit or rollback actions have not yet been performed..

The **rollback_upgrade()** method might be invoked by the user if the user issued **upgrade_object()** for several objects in a collection and would prefer to upgrade none of the objects in the collection, if all of them cannot be upgraded. The method allows an instance of an old implementation to resume processing messages.

The **revert_upgrade()** method is used after a commit to revert an instance of the new implementation of an object to an instance of the old implementation..The Upgrade Mechanisms transfer the transformed state of the instance of the new implementation to the instance of the old implementation.

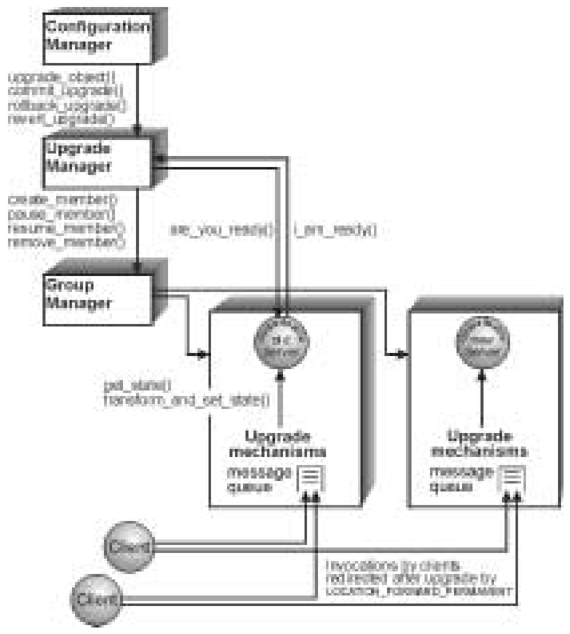


Figure 4-1 The Online Upgrade infrastructure

```
module  OnlineUpgrades {

    interface UpgradeManager {
            exception InvalidInterface {};
            exception UnknownUpgradeId {};

            void upgrade_object
                (in PortableGroup::ObjectGroup object_group,
                 in PortableGroup::TypeId type_id,
                 in PortableGroup::Location the_location,
                 in PortableGroup::FactoryInfo the_factory,
                 in boolean app_ctrl_commit)
            raises
                (PortableGroup::ObjectGroupNotFound,
                 InvalidInterface,
                 PortableGroup::NoFactory,
                 PortableGroup::ObjectNotCreated);

            void commit_upgrade
                (in PortableGroup::ObjectGroup object_group)
            raises
                (PortableGroup::ObjectGroupNotFound);

            void rollback_upgrade
                (in PortableGroup::ObjectGroup object_group)
            raises
                (PortableGroup::ObjectGroupNotFound);

            void revert_upgrade
                (in PortableGroup::ObjectGroup object_group,
                 in PortableGroup::TypeId type_id,
                 in PortableGroup::Location the_location,
                 in PortableGroup::FactoryInfo the_factory)
            raises
                (PortableGroup::ObjectGroupNotFound,
                 InvalidInterface,
                 PortableGroup::NoFactory,
                 PortableGroup::ObjectNotCreated);

            void i_am_ready
                (in upgradeId,
                 in boolean ready)
            raises
                (UnknownUpgradeId);

    };
};
```

4.2.1 Exception

exception InvalidInterface {};

The old implementation of the object and the upgraded implementation of the object do not have the same IDL interface.

4.2.2 Operations

4.2.2.1 upgrade_object

The method upgrades the instances of the old implementation of an object and instantiates the new implementation at the location specified. The method allows the application to commit the upgrade explicitly or to delegate that responsibility to the Upgrade Manager, depending on whether the value of the **app_ctrl_commit** parameter is true or false.

```
void upgrade_object
      (in PortableGroup::ObjectGroup object_group,
       in PortableGroup::TypeId type_id,
       in PortableGroup::Location the_location,
       in PortableGroup::FactoryInfo the_factory,
       in boolean app_ctrl_commit)
raises
      (PortableGroup::ObjectNotFound,
       InvalidInterface,
       PortableGroup::NoFactory,
       PortableGroup::ObjectNotCreated);
```

Parameters

object_group	The object group reference of the object-to-be-upgraded.
type_id	The type of the object-to-be-upgraded.
the_location	The location at which the upgraded implementation is to be instantiated.
the_factory	The factory that is to be used to create the instance of the upgraded implementation at the given location.
app_ctrl_commit	A boolean that allows the application to commit the upgrade explicitly or to delegate that responsibility to the Upgrade Manager.

Raises

PortableGroup::ObjectGroupNotFound if the object group for the object-to-be-upgraded is not found.

InvalidInterface if the current implementation and the upgraded implementation do not have the same IDL interface.

PortableGroup::No Factory if the factory that is to be used to create the upgraded implementation of the object is not found, or is not capable of creating an instance of the upgraded implementation.

PortableGroup::ObjectNotCreated if an instance of the upgraded implementation is not created.

4.2.2.2 *commit_upgrade*

The method allows the application to commit the upgrade explicitly.

```
void commit_upgrade
    (in PortableGroup::ObjectGroup object_group)
raises
    (PortableGroup::ObjectGroupNotFound);
```

Parameter

object_group	The object group reference for the object to be committed.

Raises

PortableGroup::ObjectGroupNotFound if the object group for the object to be committed is not found.

4.2.2.3 *rollback_upgrade*

The method allows the application to rollback the upgrade before it is committed.

```
void rollback_upgrade
    (in PortableGroup::ObjectGroup object_group)
raises
    (PortableGroup::ObjectGroupNotFound);
```

Parameter

object_group	The object group reference for the object for which the upgrade is to be rolled back.

Raises

PortableGroup::ObjectGroupNotFound if the object group for the object for which the upgrade is to be rolled back is not found.

4.2.2.4 *revert_upgrade*

The method allows the application to revert the upgraded implementation to the old implementation after it is committed. An object can be reverted only to the immediately preceding implementation, *i.e.*, the implementation from which the **upgrade_object()** method was applied to reach the current implementation.

```
void revert_upgrade
      (in PortableGroup::ObjectGroup object_group,
       in PortableGroup::TypeId type_id,
       in PortableGroup::Location the_location,
       in PortableGroup::FactoryInfo the_factory)
raises
      (PortableGroup::ObjectGroupNotFound,
       InvalidInterface,
       PortableGroup::NoFactory,
       PortableGroup::ObjectNotCreated);
```

Parameter

object_group	The object group reference for the object to be reverted.
type_id	The type of the object to be reverted.
the_location	The location at which the instance of the old implementation of the object is to be instantiated.
the_factory	The factory that is to be used to create the instance of the old implementation at the given location.

Raises

PortableGroup::ObjectGroupNotFound if the object group for the object to be reverted is not found.

InvalidInterface if the current implementation and the upgraded implementation do not have identical IDL interfaces.

PortableGroup::No Factory if the factory that is to be used to create the instance of the upgraded implementation of the object is not found or is not capable of creating an instance of the upgraded implementation.

PortableGroup::ObjectNotCreated if the instance of the new implementation of the object is not created.

4.2.2.5 i_am_ready

The method allows an upgradeable application object to inform the Upgrade Manager whether or not it is ready to be upgraded. If the **ready** parameter is equal to true, then it is in a safe and quiescent state and can be upgraded. If the **ready** parameter is equal to false, then it is not in a safe and quiescent state and so cannot be upgraded. The application object being upgraded invokes **i_am_ready()** in response to the Upgrade Manager’s invocation of **are_you_ready()**.

```
void i_am_ready
            (in unsigned long upgradeId,
             in boolean ready)
        raises
            (UnknownUpgradeId);
```

Parameter

upgradeId	The upgrade identifier of the instance of the object being upgraded.
ready	True if the instance of the object is in a safe and quiescent state and, thus, ready to be upgraded.

Raises

UnknownUpgradeId if the Upgrade identifier is not found.

Upgradeable Applications 5

5.1 Overview

This chapter defines the **Upgradeable** interface that application objects must implement if they are to be upgraded. The chapter allows defines the **PortableState** module, which contains the **Checkpointable** interface. The intention in defining the **PortableState** module is to allow its use for both Fault Tolerance and Online Upgrades and also for other specifications that require checkpointing, such as Data Parallel CORBA and Load Balancing. In particular, the **Upgradeable** interface inherits the **PortableState::Checkpointable** interface.

The **Upgradeable** interface defines the **are_you_ready()** method that the Upgrade Manager invokes on an application object instance that implements this interface. When the Upgrade Manager invokes this method, the Upgrade Manager is asking that object instance if it is in a safe and quiescent state and, therefore, is ready to be upgraded. If the object instance is in a safe and quiescent state, it invokes the **i_am_ready()** method of the Upgrade Manager with the value of **ready** equal to true; otherwise, it invokes the **i_am_ready()** method of the Upgrade Manager with the value of **ready** equal to false.

In addition to the **are_you_ready()** method, the **Upgradeable** interface contains the **get_state()** and **transform_and_set_state()** methods**.**

When an object instance is ready to be upgraded, the Upgrade Mechanisms invoke the **get_state()** operation on that instance. They then invoke the **transform_and_set_state()** operation on the instance of the new implementation of the object, using the state retrieved from the instance of the old implementation as the parameter. The **transform_and_set_state()** operation transforms the state of the instance of the old implementation into the state for the instance of the new implementation, providing values for new variables that are needed by the instance of the new implementation. The **transform_and_set_state()** operation then assigns the transformed state to the instance of the new implementation.

Similarly, on reverting an upgrade after it is committed, the Upgrade Mechanisms first invoke the **get_state()** operation of the instance of the new implementation of the object to retrieve the state of that instance. They then invoke the **transform_and_set_state()** operation on the instance of the old implementation, using the state retrieved from the instance of the new implementation as the parameter. The **transform_and_set_state()** operation transforms the state of the instance of the new implementation into the state for the instance of the old implementation, omitting any new variables and providing values for old variables that are needed by the instance of the old implementation. The **transform_and_set_state()** operation then assigns the transformed state to the instance of the old implementation.

The **get_state()** and **transform_and_set_state()** methods of the **Upgradeable** interface must be programmed by the application programmer or perhaps are generated by a source code preprocessor tool, with the help of the application programmer.

5.2 PortableState Module

The **PortableState** module contains the **Checkpointable** interface. The **PortableState::Checkpointable** interface provides **get_state()** and **set_state()** methods, which enable the state of an object to be transferred from one member of an object group to another member of the object group.

The mechanisms invoke the **get_state()** method on a member of an object group to retrieve its state from that member. They then invoke the **set_state()** method on a member of the object group to assign its state to that member.

```
module PortableState {
    typedef sequence<octet> State;

    exception NoStateAvailable {};
    exception InvalidState {};

    // Specification of the Checkpointable interface
    interface Checkpointable {
        State  get_state()
        raises
            (NoStateAvailable);

        void set_state
            (in State s)
        raises
            (InvalidState);
    };

};
```

5.2.1 Identifiers

typedef sequence<octet> State;

The state of an object.

5.2.2 Exceptions

exception NoStateAvailable {};

This exception is thrown if the state of the object is not available.

exception InvalidState {};

This exception is thrown if the state being supplied to the object is not a valid state for the object.

5.2.3 Operations

5.2.3.1 get_state

The method obtains the state of the application object on which it is invoked. The method is invoked by the underlying mechanisms. When the mechanisms invoke **get_state()** on the application object, the application object returns its state.

```
State  get_state()
raises
    (NoStateAvailable);
```

Return Value

The state of the application object on which the method is invoked.

Raises

NoStateAvailable if the state is not available.

5.2.3.2 set_state

The method sets the state of the application object on which it is invoked. The method is invoked by the underlying mechanisms. When the mechanisms invoke **set_state()**, they assign the state to the application object.

```
void set_state
    (in State s)
raises
    (InvalidState);
```

Parameters

s	The state to be used to set the state of the application object on

which the method is invoked.

Raises

InvalidState if the parameter s is not a valid state.

5.3 Upgradeable Interface

The **Upgradeable** interface must be inherited by each application object that is to be upgraded. The **Upgradeable** interface inherits the **PortableState::Checkpointable** interface, which contains the **get_state()** method. In addition, the **Upgradeable** interface defines the **are_you_ready()** and **transform_and_set_state()** methods. The **are_you_ready()** method is invoked by the Upgrade Manager. The **get_state()** and **transform_and_set_state()** methods are invoked by the Upgrade Mechanisms.

```
module OnlineUpgrades {

    // Specification of the Upgradeable interface that application objects
    // that are to be upgraded must inherit
    interface Upgradeable :
        PortableState::Checkpointable {
            boolean are_you_ready
                (unsigned long upgradeId);

            void transform_and_set_state
                (in State s)
            raises
                (InvalidState);
    };
};
```

5.3.1 Operations

5.3.1.1 are_you_ready

The method is invoked by the Upgrade Manager on the application object. When the Upgrade Manager issues the **are_you_ready()** invocation, the Upgrade Mechanisms do not deliver any further method invocations to it (except for the **get_state()** invocation described below); instead, they queue such request messages, and deliver them in due course to the instance of the upgraded implementation. If the object invokes **i_am_ready()** with the **ready** parameter equal to false, the Upgrade Manager reinvokes **are_you_ready()** after a delay defined by the PauseInterval property. If the object invokes **i_am_ready()** with the value of **ready** equal to false and the PauseInterval property has value 0, the Upgrade Manager rolls back the attempt to upgrade.

Invocation of the **are_you_ready()** method queries the instance of the old implementation about whether it is in a safe and quiescent state. A safe state may be determined by the internal state of the object and, possibly, by the state of other objects or of physical equipment being controlled by the object. A quiescent state is a state in which the object is not executing any method that has been invoked on it. The instance of the old implementation may delay the reply until it has reached a safe state.

If the object being upgraded is multithreaded, quiescence requires that all of the threads of the object are suspended. As shown in Figure 5-1, if the value of the CallBack Style property for the group is false, when the Upgrade Manager issues the **are_you_ready()** invocation, the Upgrade Mechanisms deliver no further method invocations to the old implementation (except for the **get_state()** invocation described below). The manner in which the Upgrade Mechanisms determine that all of the threads of an implementation have suspended is not defined and is vendor-specific. The manner in which the instance of the old implementation reaches a safe state, and ensures that all of its threads have suspended, is not defined and is application-specific. For many objects, the **are_you_ready()** method can be implemented by immediately invoking **i_am_ready()** with a **ready** parameter equal to true.

void are_you_ready(in unsigned long upgradeId);

Parameter

upgradeId	The upgrade identifier of the instance of the old implementation being upgraded.

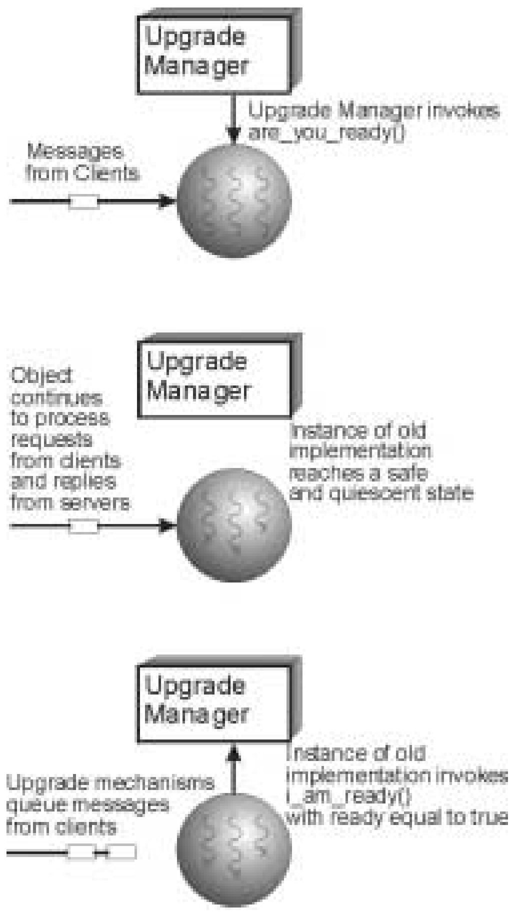

Upgrade Manager
Upgrade Manager invokes are_you_ready()
Messages from Clients
Upgrade Manager
Object continues to process requests from clients and replies from servers
Instance of old implementation reaches a safe and quiescent state
Upgrade Manager
Instance of old implementation invokes i_am_ready() with ready equal to true
Upgrade mechanisms queue messages from clients

Figure 5-1 **are_you_ready()** when the QuiescenceStyle property has the value false

If the QuiescenceStyle property for the group has the value true, as shown in Figure 5-2, the Upgrade Mechanisms continue to deliver request messages to the instance of the old implementation after the invocation of **are_you_ready()**. When the instance of the old implementation invokes **i_am_ready()** with the **ready** parameter equal to true, the Upgrade Mechanisms queue all further request messages and do not deliver them to the instance of the old implementation.

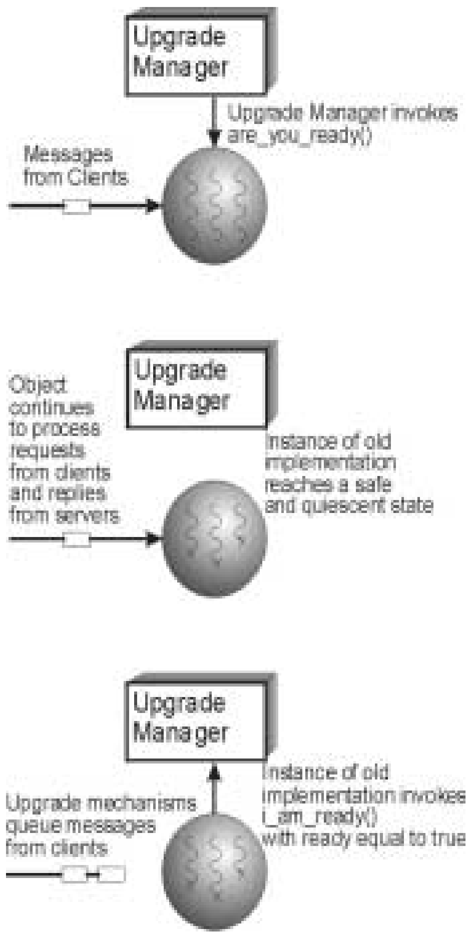

Upgrade Manager
Upgrade Manager invokes are_you_ready()
Messages from Clients
Upgrade Manager
Object continues to process requests from clients and replies from servers
Instance of old implementation reaches a safe and quiescent state
Upgrade Manager
Upgrade mechanisms queue messages from clients
Instance of old implementation invokes i_am_ready() with ready equal to true

Figure 5-2 **are_you_ready()** when the QuiescenceStyle property has the value true

5.3.1.2 *get_state*

When upgrading an object, the method retrieves the state of an instance of the old implementation of an object, which is then transformed into the state of an instance of the new implementation of the object, before it is assigned to that new instance. Similarly, when reverting an object after the upgrade has been committed, the method retrieves the state of an instance of the new implementation of an object, which is then transformed back into the state of an instance of the old old implementation of the object.

```
State get_state( )
raises
    (NoStateAvailable);
```

Return Value

The state of the instance of the implementation of the object on which the method is invoked.

Raises

NoStateAvailable if the state of the instance of the implementation of the object is not available.

5.3.1.3 transform_and_set_state

When upgrading from an instance of the old implementation of an object to an instance of the new implementation of an object, the method transforms the state of an instance of the old implementation of the object, returned by **get_state()**, into the state of the new implementation of the object and assigns the transformed state to an instance of the new implementation of the object. Similarly, when reverting from an instance of the new implementation of an object to an instance of the old implementation of an object, the method transforms the state of an instance of the new implementation of the object, returned by **get_state()**, into the state of the old implementation of the object and assigns the state to an instance of the old implementation of the object.

```
void transform_and_set_state(in State s)
raises
    (InvalidState);
```

Parameter

s	The state of an instance of the implementation of the object.

Raises

InvalidState if the state of is not valid.

Figure 5-3 shows the transfer of state from an instance of the old implementation to an instance of the new implementation, and vice versa.

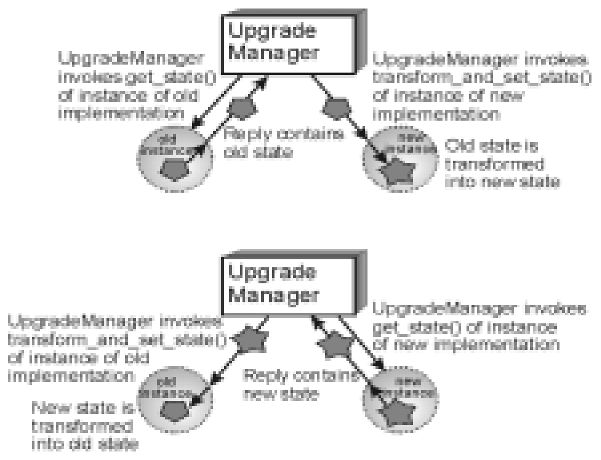

Upgrade Manager
UpgradeManager invokes get_state() of instance of old implementation
UpgradeManager invokes transform_and_set_state() of instance of new implementation
old instance
Reply contains old state
new instance
Old state is transformed into new state
Upgrade Manager
UpgradeManager invokes transform_and_set_state() of instance of old implementation
UpgradeManager invokes get_state() of instance of new implementation
old instance
Reply contains new state
new instance
New state is transformed into old state

Figure 5-3 At the top, transfer of state from an instance of the old implementation to an instance of the new implementation and, at the bottom, transfer of state from an instance of the new implementation to an instance of the old implementation

During an upgrade, the **transform_and_set_state()** method is used to transform the state of an instance of the old implementation of an object into the state of an instance of the new implementation. During a revert, the **transform_and_set_state()** method is used to transform the state of an instance of the new implementation of an object into the state of an instance of the old implementation. It cannot be expected that the old implementation already contains an appropriate **transform_and_set_state()** method because the old implementation was, presumably, written before the new implementation was devised. Consequently, the first stage of an upgrade involves upgrading the old implementation to a version of the old implementation that contains the appropriate **transform_and_set_state()** method. That version of the old implementation can then be upgraded to the new implementation.

Usage of the Specifications 6

6.1 Example Use Case

We start with an existing object that has not been upgraded previously and, thus, does not inherit the **Upgradeable** interface. We assume that the object is not replicated for fault tolerance and that the Upgrade infrastructure uses IIOP with point-to-point communication, rather than multicast group communication. The infrastructure controlled membership style is used.

1. Program the **are_you_ready()**, **get_state()** and **transform_and_set_state()** methods of the **Upgradeable** interface for the existing object implementation. Note that it is not possible to perform an online upgrade of an object that does not implement these methods of the **Upgradeable** interface.
2. Reload and restart the object, with its **Upgradeable** interface, as a singleton object group. (This is not an "online upgrade"). Publish the Interoperable Object Group Reference (IOGR). At this stage, the IOGR has only one profile, that of the old implementation of the object.

   (This setup might be considered to be a simple fault-tolerant system in which the object group has the Cold Passive Replication Style, where the primary member of the group executes and the backup members do not execute, and where the Initial Number of Replicas is one.)

We now have an Upgradeable object that we wish to upgrade.

1. Program the new upgraded implementation of the object, i.e., the application logic.
2. Program the **are_you_ready()**, **get_state()** and **transform_and_set_state()** methods for the upgraded implementation. Note that the **transform_and_set_state()** method contains the code to transform the state of the instance of the old implementation into the state of the instance of the new implementation, and vice versa.

3. Compile the new implementation of the object and store the compiled code in the implementation repository. Install a factory for the new implementation at the location where an instance of the new implementation is to be created.

4. Perform whatever testing is necessary to validate the new implementation.

As shown in Figure 6-1, higher level application software invokes the **upgrade_object()** method of the Upgrade Manager, citing the IOGR and the type of the existing implementation of the object and also the location and the factory information for the instance of the new upgraded implementation of the object. In this example, we assume that the **app_ctrl_commit** parameter of the **upgrade_object()** method invocation has the value true.

1. The Upgrade Manager now invokes the **create_member()** method of the Group Manager to create an instance of the new implementation as a member of the object group. (Essentially, the new member is created as a cold backup member, which does not execute.) The **create_member()** method returns a new IOGR that contains profiles for both the old implementation and the new implementation.

2. The Upgrade Manager invokes the **are_you_ready()** method of the instance of the old implementation and the Upgrade Mechanisms stop delivering request messages to the instance of the old implementation and start queuing those messages for the instance of the new implementation (assuming that the QuiescenceStyle property has the value false). If the instance of the old implementation determines that it is in a safe and quiescent state, it invokes the **i_am_ready()** method of the Upgrade Manager with the **ready** parameter equal to true; otherwise, it invokes the **i_am_ready()** method of the Upgrade Manager with the **ready** parameter equal to false and the Upgrade Manager rolls back the upgrade. The Upgrade Manager tries again to invoke **are_you_ready()** after a time determined by the PauseInterval property.

3. If the instance of the old implementation invokes the **i_am_ready()** method with the **ready** parameter equal to true, the Upgrade Manager invokes the **pause_member()** method of the Group Manager.

4. The Upgrade Mechanisms determine that the instance of the old implementation has quiesced with all of its threads suspended.

5. The Upgrade Mechanisms invoke the **get_state()** method of the instance of the old implementation. They record the **get_state()** message and its reply ahead of the messages that they queued following the invocation of the **are_you_ready()** method.

6. The Upgrade Mechanisms invoke the **transform_and_set_state()** method of the instance of the new implementation of the object, supplying the state of the instance of the old implementation of the object returned by **get_state()** as the parameter. The **transform_and_set_state()** method transforms the state of the instance of the old implementation into the state of the instance of the new implementation and then assigns the transformed state to the instance of the new implementation.

At this point, the **upgrade_object()** method returns.

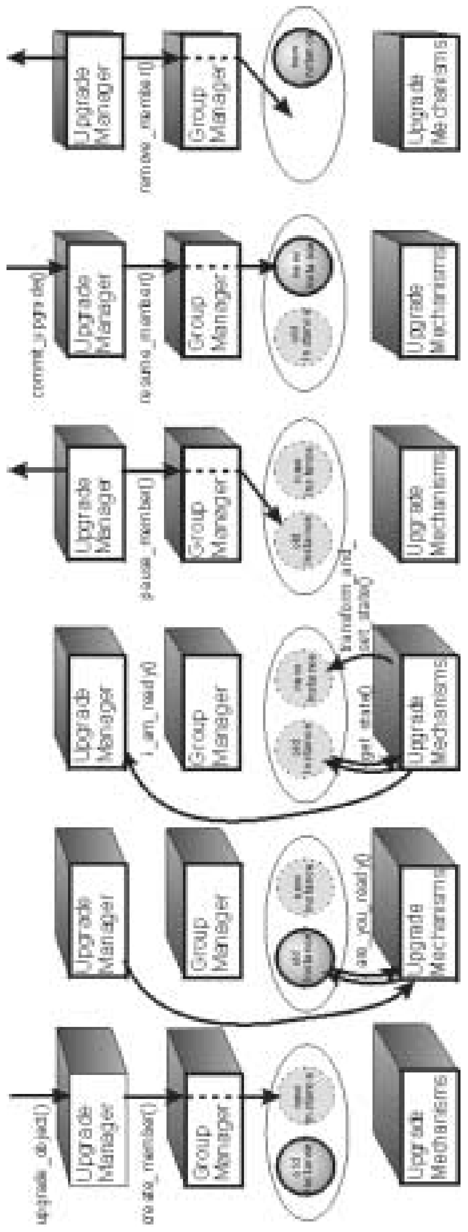


Figure 6-1 **upgrade_object()** and **commit_upgrade()**

Because the **app_ctrl_commit** parameter is set to true, the higher level application software now invokes the **commit_object()** method of the Upgrade Manager.

1. The Upgrade Manager invokes the **resume_member()** method of the Group Manager to start the instance of the new implementation processing requests. The **resume_member()** method returns a new IOGR for the instance of the new implementation.
2. For each of the messages queued for the instance of the old implementation, the Upgrade Mechanisms generate a LOCATION_FORWARD or a LOCATION_FORWARD_PERMANENT reply and transmit that reply back to the client that originated the message. These replies carry the IOGR for the instance of the new implementation.
3. At the client, the ORB retransmits the message to the instance of the new implementation. For a LOCATION_FORWARD_PERMANENT reply, the ORB replaces the previous IOGR, that addressed the instance of the old implementation, with the new IOGR that addresses the instance of the new implementation.
4. The instance of the new implementation now starts to process messages.
5. The Upgrade Manager invokes the **remove_member()** method of the Group Manager to remove the instance of the old implementation. The **remove_member()** method returns a new IOGR for the new implementation that contains only one profile. Future invocations of the instance of the old implementation of the object trigger a LOCATION_FORWARD_PERMANENT reply, carrying that new IOGR, so that the clients can update their copies of the IOGR to the new IOGR. The Upgrade Manager may delay the invocation of the **remove_member()** method for the time specified by the RemovalInterval property, against the possibility that the **revert_object()** method might be invoked. If the instance of the old implementation has not yet been removed, the **revert_object()** method does not create another instance of the old implementation and, thus, can execute more rapidly.

This completes the upgrade of the object implementation.

As shown in Figure 6-2, the higher level application software now invokes the **revert_object()** method of the Upgrade Manager, citing the IOGR and the type of the instance of the new implementation and also the location and the factory for the instance of the old implementation. The **revert_object()** method does not have an **app_ctrl_commit** parameter because, on reversion of an upgrade, the Upgrade Mechanisms always perform the commit and, thus, such a parameter would always have the value false.

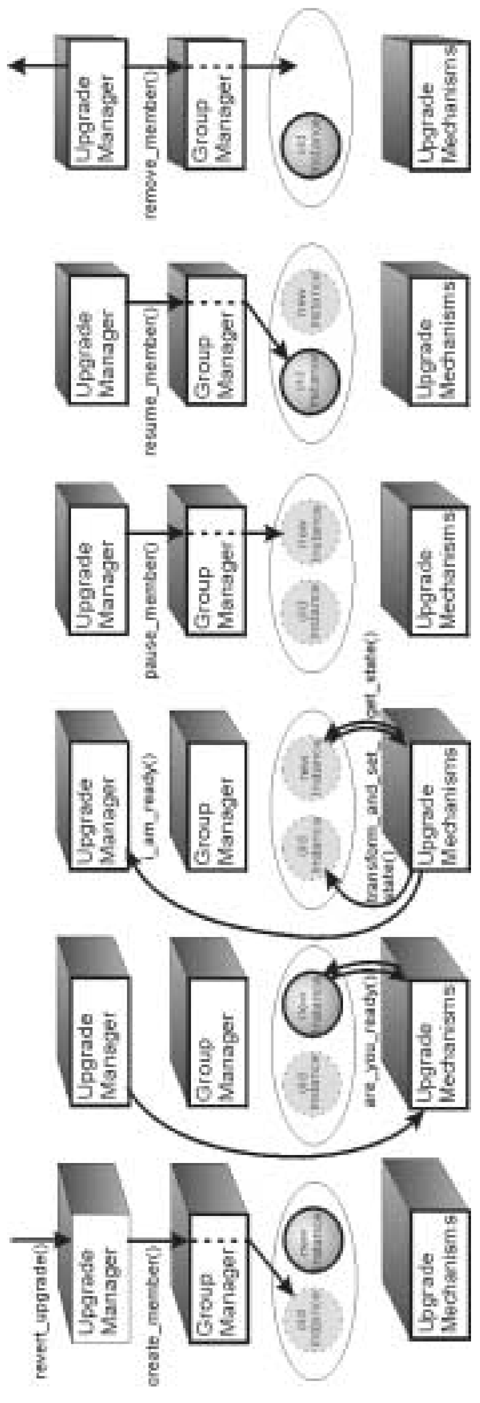


Figure 6-2 revert_upgrade()

1. The Upgrade Manager invokes the **create_member()** method of the Group Manager to create the instance of the old implementation as a member of the object group. The **create_member()** method returns a new IOGR that contains profiles for both the new implementation and the old implementation.
2. The Upgrade Manager invokes the **are_you_ready()** method of the instance of the new implementation. The Upgrade Mechanisms stop delivering request messages to the instance of the new implementation and, instead, start queuing those messages (assuming that the QuiescenceStyle property has the value false).
3. If the instance of the new implementation determines that it is in a safe and quiescent state to perform an upgrade, it invokes the **i_am_ready()** method of the Upgrade Manager with the value of **ready** equal to true. Otherwise, it invokes **i_am_ready()** with the value of **ready** equal to false and the Upgrade Manager rolls back to the instance of the old implementation and resumes that instance. The Upgrade Manager tries to invoke **are_you_ready()** again after waiting the time defined by the PauseInterval property.
4. The Upgrade Mechanisms determine that the instance of the new implementation has quiesced with all of its threads suspended.
5. The Upgrade Mechanisms invoke the **get_state()** method of the instance of the new implementation. The Upgrade Mechanisms record the message containing the **get_state()** invocation and its reply ahead of the messages that they queued when they received the **i_am_ready()** invocation with the **ready** parameter equal to true.
6. The Upgrade Manager invokes the **resume_member()** method of the Group Manager to start the instance of the old implementation processing requests. The **resume_member()** method returns a new IOGR for the instance of the old implementation.
7. The Upgrade Mechanisms invoke the **transform_and_set_state()** method of the instance of the new implementation of the object, supplying the state returned by the **get_state()** method as a parameter. The **transform_and_set_state()** method transforms the state of the instance of the new implementation into the state of the instance of the old implementation and then assigns the transformed state to the instance of the old implementation.
8. For each of the request messages queued for the instance of the new implementation, the Upgrade Mechanisms generate a LOCATION_FORWARD or LOCATION_FORWARD_PERMANENT reply and transmit that reply back to the client that originated the message. These replies carry the IOGR for the instance of the old implementation.
9. At the client, the ORB replaces the previous IOGR, that address the instance of the new implementation, with the new IOGR that addresses the old implementation, and retransmits the message to the instance of the old implementation.
10. The instance of the old implementation now starts to process messages.
11. The Upgrade Manager invokes the **remove_member()** of the Group Manager to remove the instance of the new implementation. The **remove_member()** method returns an IOGR for the old implementation, carrying only one profile. Future

invocations of the instance of the new implementation of the object trigger a LOCATION_FORWARD_PERMANENT reply, carrying the IOGR that addresses the old implementation, so that the clients can update their copies of the IOGR to address the instance of the old implementation.

This completes the reversion of an instance of the upgraded implementation to an instance of the old implementation.

Responses to RFP Requirements 7

7.1 Resolution of RFP Mandatory Requirements

6.1. Proposals shall specify interfaces to manage the upgrading of the implementation of one or more CORBA objects.

The proposed specifications for Online Upgrades provide interfaces to manage the upgrading of the implementation of a single CORBA object instance. The proposed specifications allow these interfaces to be used to upgrade multiple CORBA object instances, but provide minimal mechanisms to coordinate the upgrading of multiple CORBA object instances. Extensions to the proposed specifications could provide better coordination for the upgrading of multiple object instances.

The proposed specification exploits the notion of object groups, which was introduced by the Fault Tolerant CORBA standard. For Fault Tolerant CORBA, object groups are homogeneous in that the members of an object group are replicas. For Online Upgrades, object groups are heterogeneous in that instances of both the old implementation of an object and and the new implementation of an object are members of the object group, while the upgrade is taking place. Heterogeneous object groups can perhaps also be used to define composite objects that can be upgraded as a unit. In our opinion, the definition of compositions of multiple objects is a system structuring topic that should be addressed directly rather than as an incidental consequence of a specification that addresses primarily a different topic. Consequently, the proposed specifications do not address fully the upgrading of multiple CORBA objects.

6.2. Proposals shall specify interfaces to form and manage object groups to facilitate upgrading.

The definition of an object group, containing instances of both the original and the upgraded implementations of an object, facilitates the upgrading of an object, and also the addressing of an upgraded object through the use of the Interoperable Object Group Reference (IOGR), which the Fault Tolerant CORBA standard introduced.

The proposed specifications for Online Upgrades include a **PortableGroup** module that provides a consistent basis for the specification of object groups within several adopted or proposed specifications, including Fault Tolerant CORBA, Unreliable Multicast, Data Parallel CORBA, Load Balancing and Online Upgrades. Such an interface has already been defined by the Unreliable Multicast specifications, based on an earlier version of Fault Tolerant CORBA, rather than on the publicly available Fault Tolerant CORBA standard on which the **PortableGroup** module defined here is based.

6.3. Proposals shall address upgrades for both active and passive object groups.

The proposed specifications for Online Upgrades are orthogonal to active and passive replication, and permit the upgrading of objects that use either replication style.

6.4. Proposals shall address upgrades for both application-controlled and infrastructure-controlled object group membership styles.

The proposed specifications for Online Upgrades permit the upgrading of objects that use either application-controlled or infrastructure-controlled membership styles, through the **PortableGroup** module.

6.5.Proposals shall define how an Interoperable Object Reference, held by a client, can be used to invoke a method of the old version of the object before the object is upgraded, and of the new version of the object after the object is upgraded.

The proposed specifications for Online Upgrades exploit the Interoperable Object Group Reference as a means of addressing an object and invoking a method of the object before it is upgraded, and of the object after it is upgraded, while allowing the client object to continue to use the same reference to address the object.

All objects that inherit the **Upgradeable** interface are accessed by an Interoperable Object Group Reference. The LOCATION_FORWARD_PERMANENT reply allows an Interoperable Object Group Reference held by a client object, that addresses an object being upgraded, to be adjusted to address an instance of the upgraded implementation of the object rather than an instance of the original implementation of the object, without requiring any action by the client object.

Version Management is a topic distinct from Online Upgrades. The proposal does not address Version Management; a future OMG RFP might address Version Management.

6.6. Proposals shall address several redundancy selection strategies for active object groups, other than first arrival.

The proposed specifications for Online Upgrades are orthogonal to redundancy selection strategies and, thus, do not define redundancy selection strategies (see Section 7.2 below).

6.7. Proposals shall specify interfaces to assist with state synchronization between the old and the new versions of an object during the upgrade process.

The **PortableState** and **Upgradeable** interfaces specified in Chapter 5 define methods for synchronizing the states of an instance of the old implementation and an instance of the upgraded implementation of an object.

7.2 Resolution of RFP Optional Requirements

7.1. Proposals may specify interfaces to allow the history of the various members of an object group to be collected for off-line comparison.

The proposed specifications for Online Upgrades do not define interfaces to support the collection of a history of messages of the various members of an object group. The reasons for not including such interfaces in the proposed specifications are discussed in Section 7.2 below.

7.2. Proposals may specify interfaces to allow one of the members of an object group to be treated as an object-under-test, whose output is ignored, but whose history is recorded.

The proposed specifications for Online Upgrades do not specify interfaces to support the testing of the upgraded implementation of an object.

The scenario that leads to the request for such an interface, as a part of an Online Upgrades specification, is as follows:

- Operation of the system with only an instance of the old implementation of the object.
- Introduction of an instance of the new implementation of the object and transfer of the state of an instance of the old implementation of the object to an instance of the new implementation of the object.
- Continued operation of the system with an instance of the old implementation of the object, together with concurrent operation of an instance of the new implementation of the object as an object-under-test.
- Manual (or automatic) observation of the object-under-test, using a mechanism for the collection of messages, leading to permission to perform the upgrade.
- Switchover to operation of the system with an instance of the upgraded implementation of the object.

It is our opinion that this manner of upgrading is inappropriate.

Employing an object-under-test is a useful strategy, and it is important to conduct such testing before performing an upgrade. But object-under-test presents several substantial difficulties. The object-under-test is operating "open loop," with its outputs being ignored. If the object-under-test behaves differently from the operational implementation, the requests that are invoked on it, and the replies that it receives, are appropriate for the operational implementation, rather than for the somewhat different object-under-test. Thus, the object-under-test can accumulate error values and discrepancies that would not have accumulated during normal operation. Such error values and discrepancies might be regarded as inevitable and might be disregarded during testing, but are highly undesirable during real operation.

Consequently, it is our opinion that it is inappropriate to perform an upgrade to an implementation that has been, potentially, degraded by a period of operation under test. Rather, we advocate a scenario such as:

- Operation of the system with only an instance of the old implementation of the object.
- Introduction of an instance of the new implementation of the object and transfer of state from the instance of the old implementation to an instance of the new implementation.
- Continued operation of the system with an instance of the old implementation of the object, together with concurrent operation of an instance of the new implementation as an object-under-test.
- Manual (or automatic) observation of the object-under-test, possibly using a mechanism to collect a history of messages. Such observation might lead to permission to perform the upgrade.
- Termination (or continued operation) of the object-under-test.
- Introduction of a "fresh" new implementation of the object and transfer of state from an instance of the old implementation of the object to an instance of the "fresh" new implementation of the object.
- Immediate switchover to the new implementation of the object in the operational system.

It is our assessment that this scenario allows observation of an object-under-test to confirm that it can operate correctly while minimizing the risk that the new implementation has been degraded by "open loop" operation prior to the switchover.

Consequently, it is our opinion that the specification of interfaces for Online Upgrades and the specification of interfaces for online testing can be separated, although their implementations might have some common mechanisms. Furthermore, other individuals are more capable of devising appropriate interfaces for online testing than we are, and thus, we do not define interfaces for online testing in this proposal.

7.3. Proposals may specify interfaces that support control over acceptance of an upgraded object implementation, allowing that implementation to execute in the operational system, or of rejection of an upgrade, causing fallback to the prior version of the object.

The proposed specifications for Online Upgrades provide control, at two levels, over the acceptance or rejection of a new implementation of an object.

During the upgrade, after an instance of the new implementation of the object has been installed but before the switchover from an instance of the old implementation to an instance of the new implementation, the operator or management software can invoke "Commit" to perform the switchover, or "Rollback" to resume operation using the instance of the old implementation of the object. The instance of the old implementation continues with its current state and processes all of the incoming messages, incurring only a short pause in operation. The instance of the new implementation processes no messages. This mechanism is most useful for upgrading several objects together, where the reason for rolling back the upgrade is that the request to upgrade one of the objects was rejected with an error reply, for example, because of a signature inconsistency.

After the switchover and during operation of the instance of the new implementation of the object, the operator or management software can invoke "Revert" to return to using the old implementation. A "Revert" is essentially equivalent to an "upgrade" from the new implementation to the old implementation of the object, and involves the same steps and mechanisms (and overheads) as a normal upgrade. The state from which the instance of the old implementation resumes operation is the then current state of the instance of the new implementation of the object transformed into the state of the old implementation.

7.4. Proposals may address the upgrading of the format of the persistent state of an object.

The proposed specifications for Online Upgrades do not address the upgrading of the format of the persistent state of an object.

7.5. Proposals may address the issue of a Platform Independent Model.

A Platform Independent Model for Online Upgrades is provided in Section 2.9.

7.3 Responses to RFP Issues

8.1 Proposals should discuss how the fallback of a new version of an object to its old version will occur.

The proposed specifications for Online Upgrades address, in Section 4.2, how fallback (**rollback_upgrade()** and **revert_upgrade()**) of an instance of an upgraded implementation to an instance of the old implementation will occur.

8.2 Proposals should discuss how a reply can be associated with the replying object, such as version identification and control issues.

The proposed specifications for Online Upgrades address how a reply can be associated with the replying object in Section 2.6. The proposed specifications do not provide a version control system. While a basic version control system might involve only a simple sequential count, sophisticated version control systems are complex and non-linear. It is our assessment that such a version control system would be implemented most appropriately as a part of a higher level management system, and should be defined by a separate specification.

The Interoperable Object Group Reference (IOGR) does contain a simple version number for the IOGR. Using the IOGR version number, it is possible for a higher level management function to determine, from its internal databases, the fully elaborated version designation of an object.

8.3 Proposals should discuss security and access control issues that go beyond the existing CORBAsecurity model.

The proposed specifications for Online Upgrades do not address security and access control issues, which we regard as issues orthogonal to these specifications.

Compliance and Conformance 8

8.1 Mandatory and Optional Interfaces

The infrastructure-controlled MembershipStyle (MEMB_INF_CTRL) is mandatory. The application-controlled MembershipStyle (MEMB_APP_CTRL) and the application-controlled Commit (**app_ctrl_commit**) are optional. If application control is not supported, the MembershipStyle must be set to MEMB_INF_CTRL and the **app_ctrl_commit** parameter must be set to false. All of the other interfaces defined is this proposal are mandatory.

8.2 Proposed Compliance Points

Two compliance points are defined. The first compliance point implements the membership style MEMB_INF_CTRL, but does not implement the membership style MEMB_APP_CTRL. The second compliance point implements both membership styles, MEMB_INF_CTRL and MEMB_APP_CTRL.

8.3 Changes to Existing Specifications

The proposed specifications define a **PortableGroup** module and also a **PortableState** module.

The **PortableGroup** module is derived from the publicly available specification for Fault Tolerant CORBA. It differs slightly from the **PortableGroup** module that is defined by the Unreliable Multicast specification, which was based on an earlier version of Fault Tolerant CORBA. Other adopted and proposed specifications, namely Fault Tolerant CORBA, Unreliable Multicast, Data Parallel CORBA and Load Balancing, could be modified to make use of the **PortableGroup** module defined here.

The **PortableState** module derives from the publicly available specification for Fault Tolerant CORBA. The Fault Tolerant CORBA specification could be modified to make use of the **PortableState** module. Other adopted and proposed specifications, such as Fault Tolerant CORBA, could be modified to make use of the **PortableState** module defined here.

Consolidated IDL 9

```
#ifndef _PortableGroup_IDL_
#define _PortableGroup_IDL_

#include “CosNaming.idl”          // from 98-10-19.idl
#include “IOP.idl”                // from 98-03-01.idl
#include “GIOP.idl”               // from 98-03-01.idl
#include “CORBA.idl”              // from 98-03-01.idl

#pragma prefix “omg.org”

module PortableGroup {

    // Specification for Interoperable Object Group References
    typedef string DomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;

    struct TagGroupTaggedComponent { // tag = TAG_PG_GROUP;
            GIOP::Version              version;
            DomainId                   domain_id;
            ObjectGroupId              object_group_id;
            ObjectGroupRefVersion      object_group_ref_version;
    };

    // Specification of Common Types and Exceptions for Group Management
    interface GenericFactory;

    typedef CORBA::RepositoryId TypeId;
    typedef Object ObjectGroup;

    typedef string Name;
    typedef any Value;

    struct Property {
            Name nam;
            Value val;
    };
    typedef sequence<Property> Properties;

    typedef CosNaming::Name Location;
    typedef sequence<Location> Locations;
    typedef Properties Criteria;

    struct FactoryInfo {
            GenericFactory the_factory;
            Location the_location;
            Criteria the_criteria;
    };
    typedef sequence<FactoryInfo>  FactoryInfos;

    typedef unsigned short MembershipStyleValue;
```

```
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;

typedef FactoryInfos FactoriesValue;

exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception UnsupportedProperty {
                Name nam;
};
exception InvalidPropertyValue {
                Name nam;
                Value val;
};
exception NoFactory {
               Location the_location;
               TypeId type_id;
};
exception InvalidCriteria {
               Criteria invalid_criteria;
};
exception CannotMeetCriteria {
                Criteria unmet_criteria;
};

// Specification of the PropertyManager interface
interface PropertyManager {

       void set_default_properties
              (in Properties props)
        raises
              (InvalidPropertyValue,
               UnsupportedProperty);

       Properties get_default_properties();

       void remove_default_properties
              (in Properties props)
       raises
              (InvalidPropertyValue,
               UnsupportedProperty);

       void set_type_properties
              (in TypeId type_id,
               in Properties overrides)
       raises
```

```
        (InvalidPropertyValue,
         UnsupportedProperty);

    Properties get_type_properties
        (in TypeId type_id);

    void remove_type_properties
        (in TypeId type_id,
         in Properties props)
    raises
        (InvalidPropertyValue,
         UnsupportedProperty);

    void set_properties_dynamically
        (in ObjectGroup object_group,
         in Properties overrides)
    raises
        (ObjectGroupNotFound,
         InvalidPropertyValue,
         UnsupportedProperty);

    Properties get_properties
        (in ObjectGroup object_group)
    raises
        (ObjectGroupNotFound);
};

// Specification of the ObjectGroupManager interface
interface ObjectGroupManager {

    ObjectGroup create_member
        (in ObjectGroup object_group,
         in Location the_location,
         in TypeId type_id,
         in Criteria the_criteria)
    raises
        (ObjectGroupNotFound,
         MemberAlreadyPresent,
         NoFactory,
         ObjectNotCreated,
         InvalidCriteria,
         CannotMeetCriteria);

    ObjectGroup add_member
        (in ObjectGroup object_group,
         in Location the_location,
         in Object member)
    raises
        (ObjectGroupNotFound,
         MemberAlreadyPresent,
         ObjectNotAdded);
        ObjectGroup remove_member
                (in ObjectGroup object_group,
                 in Location the_location)
        raises
                (ObjectGroupNotFound,
                 MemberNotFound);

        Locations locations_of_members
                (in ObjectGroup object_group)
        raises
                (ObjectGroupNotFound);

        ObjectGroupId get_object_group_id
                (in ObjectGroup object_group)
        raises
                (ObjectGroupNotFound);

        ObjectGroup get_object_group_ref
                (in ObjectGroup object_group)
        raises
                (ObjectGroupNotFound);

        Object get_member_ref
                (in ObjectGroup object_group,
                 in Location the_location)
        raises
                (ObjectGroupNotFound,
                 MemberNotFound);
};

// Specification of the GenericFactory interface
interface GenericFactory {
        typedef any FactoryCreationId;

        Object create_object
                (in TypeId type_id,
                 in Criteria the_criteria,
                 out FactoryCreationId factory_creation_id)
        raises
                (NoFactory,
                 ObjectNotCreated,
                 InvalidCriteria,
                 InvalidPropertyValue,
                 CannotMeetCriteria);

        void delete_object
                (in FactoryCreationId factory_creation_id)
        raises
                (ObjectNotFound);
};
```

```
};
#endif          // for #ifndef _PortableGroup_IDL_
```

```
#ifndef _PortableState_IDL_
#define _PortableState_IDL_

#include “CosNaming.idl”          // from 98-10-19.idl
#include “IOP.idl”                // from 98-03-01.idl
#include “GIOP.idl”               // from 98-03-01.idl
#include “CORBA.idl”              // from 98-03-01.idl

#pragma prefix “omg.org”

module PortableState {
    typedef sequence<octet> State;

    exception NoStateAvailable {};
    exception InvalidState {};

    // Specification of the Checkpointable interface
    interface Checkpointable {
            State  get_state()
            raises
                (NoStateAvailable);

            void set_state
                (in State s)
            raises
                (InvalidState);
    };
};
#endif         // for #ifndef _PortableState_IDL_
```

```
#ifndef _OnlineUpgrades_IDL_
#define _OnlineUpgrades_IDL_

#include “CosNaming.idl”          // from 98-10-19.idl
#include “IOP.idl”                // from 98-03-01.idl
#include “GIOP.idl”               // from 98-03-01.idl
#include “CORBA.idl”              // from 98-03-01.idl

#pragma prefix “omg.org”

module OnlineUpgrades {
    // Specification of types for upgrade properties
    typedef boolean QuiescenceStyleValue;
    typedef long PauseIntervalValue;
    typedef long RemovalIntervalValue;

    // Specification of the GroupManager interface
    interface GroupManager :
        PortableGroup::PropertyManager,
        PortableGroup::ObjectGroupManager,
        PortableGroup::GenericFactory {

            void pause_member
                (in ObjectGroup object_group,
                 in Location the_location)
            raises
                (MemberNotFound);

            ObjectGroup resume_member
                (in ObjectGroup object_group,
                 in Location the_location)
            raises
                (MemberNotFound);
    };

    // Specification of the UpgradeManager interface
    interface UpgradeManager {
            exception InvalidInterface {};
            exception UnknownUpgradeId {};

            void upgrade_object
                (in PortableGroup::ObjectGroup object_group,
                 in PortableGroup::TypeId type_id,
                 in PortableGroup::Location the_location,
                 in PortableGroup::FactoryInfo the_factory,
                 in boolean app_ctrl_commit)
            raises
                (PortableGroup::ObjectNotFound,
                 InvalidInterface,
                 PortableGroup::NoFactory,
                 PortableGroup::ObjectNotCreated);
            void commit_upgrade
                (in PortableGroup::ObjectGroup object_group)
            raises
                (PortableGroup::ObjectGroupNotFound);

            void rollback_upgrade
                (in PortableGroup::ObjectGroup object_group)
            raises
                (PortableGroup::ObjectGroupNotFound);

            void revert_upgrade
                (in PortableGroup::ObjectGroup object_group,
                 in PortableGroup::TypeId type_id,
                 in PortableGroup::Location the_location,
                 in PortableGroup::FactoryInfo the_factory)
            raises
                (PortableGroup::ObjectGroupNotFound,
                 InvalidInterface,
                 PortableGroup::NoFactory,
                 PortableGroup::ObjectNotCreated);

            void i_am_ready
                (in unsigned long upgradeId,
                 in boolean ready)
            raises
                (UnknownUpgradeId);
    };

    // Specification of the Upgradeable interface that application objects
    // that are to be upgraded must inherit
    interface Upgradeable :
            PortableState::Checkpointable {
                void are_you_ready
                    (in unsigned long upgradeId);

                void transform_and_set_state
                    (in State s)
                raises
                    (InvalidState);
    };
};
#endif          // for #ifndef _OnlineUpgrades_IDL_
```