# General Design Principles 2

This chapter discusses the principles that were considered in designing Object Services and their interfaces. It also addresses dependencies between Object Services, their relationship to CORBA, and their conformance to existing standards.

## 2.1 Service Design Principles

### 2.1.1 Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

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

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

### 2.1.2 Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

### 2.1.3 Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

### 2.1.4 Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

### 2.1.5 Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

### 2.1.6 Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. This is shown graphically in Figure 2-1.

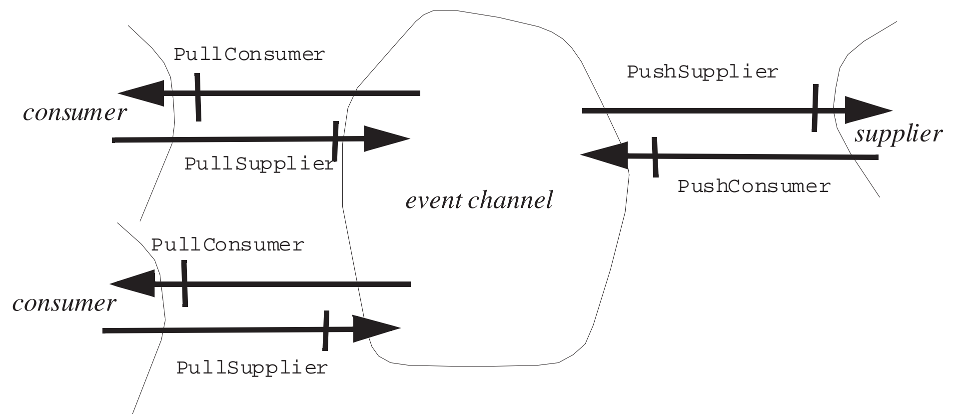


*Figure 2-1* An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

The graphical notation shown in Figure 2-1 is used throughout this document and in the full service specifications. An arrow with a vertical bar is used to show that the target object supports the interface named below the arrow and that clients holding an object reference to it of this type can invoke operations. In shorthand, one says that the object reference (held by the client) supports the interface. The arrow points from the client to the target (server) object.

A blob (misshapen circle) delineates a conspiracy of one or more objects. In other words, it corresponds to a conceptual object that may be composed of one or more CORBA objects that together provide some coordinated service to potentially multiple clients making requests using different object references.

### 2.1.7 Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms

### 2.1.8 Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

### 2.1.9 Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

## 2.2 Interface Style Consistency

### 2.2.1 Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

### 2.2.2 Explicit Versus Implicit Operations

Operations are always explicit rather than implied e.g. by a flag passed as a parameter value to some "umbrella" operation. In other words, there is always a distinct operation corresponding to each distinct function of a service.

### 2.2.3 Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by "normal" clients.

## 2.3 Key Design Decisions

### 2.3.1 Naming Service: Distinct from Property and Trading Services

The Naming Service is addressed separately from property and trading services.

Naming contexts have some similarity to property lists (that is, lists of values associated with objects though not necessarily part of the object's state). The Naming Service in general also has elements in common with a trading service. However, following the "Bauhaus" principle of keeping services as simple and as orthogonal as possible, these services have been kept distinct and are being addressed separately.

### 2.3.2 Universal Object Identity

The services described in this manual do not require the concept of object identity.

## 2.4 Integration with Future Object Services

This section discusses how the Object Services could evolve to integrate with future services, such as:

- Archive
- Backup/Restore
- Change Management (Versioning)
- Data Interchange
- Implementation Repository
- Internationalization
- Logging
- Recovery
- Replication
- Startup

### 2.4.1 Archive Service

**Persistent Object Service.** The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service should be able to archive objects stored with the Persistent Object Service.

**Externalization Service.** The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service could use the Externalization Service to get the internal state of objects for saving and to subsequently recreate objects with this stored state. If only persistent objects need to be archived, then the Object Persistence Service could be used instead.

### 2.4.2 Backup/Restore Service

**Externalization Service.** The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Object Externalization Service as an underlying mechanism for objects regardless of whether they are persistent.

**Persistent Object Service.** The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Persistent Object Service as an underlying mechanism for persistent objects.

**Transaction Service**. The permanence of effect property of a transaction implies that the state established by the commitment of a transaction will not be lost. To guarantee this property, the storage media on which the objects updated by the transaction are stored must be backed-up to secondary storage to ensure that they are not lost should the primary storage media fail. Similarly, the storage media used by the logging service must be restorable should the media fail. Since there are multiple components which require backup services, a single interface would be advantageous.

### 2.4.3 Change Management Service

**Persistent Object Service.** The Change Management Service supports the identification and consistent evolution of objects including version and configuration management. This service should work with the Persistent Object Service to allow persistent objects to evolve from the old to new versions.

### 2.4.4 Data Interchange Service

**Persistent Object Service**. The Data Interchange Service enables objects to exchange some or all of their associated state. This service should work with Persistent Object Service to allow state to be exchanged when one or more of the objects are persistent.

### 2.4.5 Internationalization Service

**Naming Service.** Naming Service interfaces may also need to be extended (for example, the structure of names extended, additional name resolution operations added) to better support representing and resolving names for some languages and cultures.

### 2.4.6 Implementation Repository

**Persistent Object Service**. The Implementation Repository supports the management of object implementations. The Persistent Object Service may depend on this to determine what persistent data an object contains. This dependency is at the implementation level.

### 2.4.7 Interface Repository

**Persistent Object Service**. The Interface Repository supports runtime access to OMG IDL-specified definitions such as object interfaces and type definitions. The Persistent Object Service depends on this to determine if a persistent object supports certain interfaces.

### 2.4.8 Logging Service

**Transaction Service.** A logging service implements the abstract notion of an infinitely long, sequentially-accessible, append-only file. It typically supports multiple log files, where each log file consists of a sequence of log records. New log records are written to the end of a log file, old log records can be read from any position in the file. To stop log files from growing too large for the underlying storage medium, a log service must provide an operation to archive old log records to allow the log file to be truncated.

Various components of a transaction processing system may require the services of a log service:

- Transaction Service: during the two-phase commit protocol the Transaction Service must log its state to ensure that the outcome of the committing transaction can be determined should there be a failure.
- Recoverable (transactional) objects: a log can be used to record old and new versions of a recoverable object for the purposes of supporting recovery.
- Locking service: a log can be used to record the locks held on an object at prepare time to facilitate recovery.

Since there are multiple components within a distributed transaction processing system that require the services of a log service, a single log service interface (and potentially server) that is shared between the components is clearly advantageous.

The correctness of a transaction service depends upon the services of a log service, for this reason, the log service must meet the following requirements:

1. Restart.

   A restart facility allows rapid recovery from the cold start of an application. The recovery service used by the application (indirectly through the application's use of recoverable objects) would use the restart facility to establish a *checkpoint*: a consistent point in the execution state of the application from which the recovery process can proceed. In the absence of a checkpoint the recovery service would have to scan the entire log to ensure restart recovery occurs correctly.

2. Buffering and forcing operations.

   A log service should provide two classes of operation for writing log records:

   a. An operation to buffer a log record (the record is not written directly to the underlying storage medium). Used during the execution of a transaction. Since the log record is buffered the write is inexpensive.

   b. An operation to force a log record to the underlying storage medium. Used during the two-phase commit protocol to guarantee the correctness of the transaction. Forcing a log record also flushes all previously written, but buffered, log records.

3. Robustness.

   The log service should ensure the consistency of the underlying storage medium in which log files are stored. This usually involves the log service employing protocols that update the storage in a manner that would not result in the loss of any existing data (i.e. careful updates), along with support for mirroring the storage media to tolerate media failures.

4. Archival.

   A log service should provide support for archiving log records. Archival is necessary to allow the log to be truncated to ensure that it does not grow without bounds.

5. Efficiency.

   Since the log service may be written to by multiple components within a transaction, the addition of log records must be efficient to avoid the bandwidth of log from becoming a bottleneck in the system.

### *2.4.9 Recovery Service*

**Transaction Service.** As recoverable objects are updated during a transaction, they (as resource managers) keep a record of the changes made to their state that is sufficient to undo the updates should the transaction rollback. The component responsible for this task is termed the recovery service. Various different forms of recovery are possible, however the most common form is called value logging and involves the recoverable object recording both the old and new values of the object. When a transaction is recovered due to failure, the old value of an object is used to undo changes made to the object during the transaction. Most recovery services employ the services of a logging service (described in this section) to maintain the “undo” information. The definition of a standard recovery service interface is one possible additional CORBA-compliant object service.

### *2.4.10 Replication Service*

**Persistent Object Service**. The Replication Service provides explicit replication of objects in a distributed environment and manages the consistency of replicated copies. This service could use the Persistent Object Service to manage persistent replicas.

### 2.4.11 Startup Service

**Persistent Object Service**. The Startup Service supports bootstrapping and termination of the Persistent Object Service.

### 2.4.12 Data Interchange Service

**Externalization Service.** The Data Interchange Service enables objects to exchange some or all of their associated state. This service could use the Object Externalization Service to allow state to be exchanged regardless of whether the objects are persistent.

## 2.5 Service Dependencies

The interface designs of all the services are general in nature and do not presume or require the existence of specific supporting software in order to implement them. An implementation of the Name Service, for instance, could use naming or directory services provided in a general-purpose networking environment. For example, an implementation may be based on the naming services provided by ONC or DCE. Such an implementation could provide enterprise-wide naming services to both object-based and non-object-based clients. Object-based software would see such services through the use of NamingContext objects.

Although the Object Services do not depend upon specific software, some dependencies and relationships do exist between services.

### 2.5.1 Event Service

The Event Service does not depend upon other services.

### 2.5.2 Life Cycle Service

Interfaces for the Life Cycle Service depend on the Naming Service.

The Life Cycle Service also defines compound operations that depend on the Relationship Service for the definition of object graphs. Appendix A describes the topic of compound life cycle, and its dependence on the Relationship Service, in detail.

### 2.5.3 Persistent Object Service

The Externalization Service provides functions that provide for the transformation of an object into a form suitable for storage on an external media or for transfer between systems. The Persistent Object Service uses this service as a POS protocol.

The Life Cycle Service provides operations for managing object creation, deletion, copy and equivalence. The Persistent Object Service depends on this service for creating and deleting all required objects.

The Naming Service provides mappings between user-comprehensible names and CORBA object references. The Persistent Object Service depends on this service to obtain the object reference of, say, a PDS from its name or id.

### 2.5.4 Relationship Service

The Relationship Service does not depend on other services. Note especially that the Relationship Service does not depend on any common storage service.

For guidelines about when to use the Relationship Service and when to use CORBA object references, refer to the section "The Relationship Service vs CORBA Object References," in Chapter 9.

### 2.5.5 Externalization Service

The Externalization Service works with the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. Specifically, this service uses the Life Cycle Service to create and remove Stream and StreamFactory objects. ORB services may be used in Stream implementations to identify InterfaceDef and ImplementationDef objects corresponding to an externalized object, and to support finding an appropriate factory for recreating that object at internalization time.

The Externalization Service can also work with the Relationship Service. Implementations of Stream and StreamIO operations could use the Relationship Service to ensure that multiple references to the same object or circular references don't result in duplication of objects at internalization time or in the external representation.

In addition, the Externalization Service adds compound externalization semantics to the containment and reference relationships in the Relationship Service. Detailed information is provided in "Specific Externalization Relationships" on page 8-26.

### 2.5.6 Transaction Service

As concurrent requests are processed by an object a mechanism is required to mediate access. This is necessary to provide the transaction property of isolation. The Concurrency Control Service is one possible implementation of a locking service.

The Transaction Service depends upon the Concurrency Control Service in the following ways:

- Concurrency Control Service must support transaction duration locks, which provide isolation of concurrent requests by different transactions.
- Concurrency Control Service must record transaction duration locks on persistent media, such as a log, as part of the prepare phase of commitment.
- If nested transactions are supported by the Transaction Service then the Concurrency Control Service must also support locks that provide isolation between siblings in a transaction family and provide inheritance of locks owned by a subransaction to its parent when the subtransaction commits.

- Transactional clients of the Concurrency Control Service are responsible for ensuring that all locks held by a transaction are dropped after all recovery or commitment operations have taken place. The drop-licks operation is provided by the LockCoordinator interface for this purpose.

The Transaction Service supports atomicity and durability properties through the Persistent Object Service (POS). The Transaction Service can work with the POS to support atomic execution of operations on persistent objects. Transactions and persistence are not provided by the same service. When coordination of multiple state changes are required to persistent data, a persistence service requires a transaction service. The POS can provide persistence, but its implementation needs to be changed to support transactional behavior. There are no changes to the interfaces of the POS to support transactions. The following discussion applies to support of persistence when a transaction service is required.

Support for persistence can be built from other specialized services that can also be shared by other object services. Examples include:

- Recovery service: this supports the atomicity and durability properties.
- Logging service: this is used by the recovery service to assist in supporting the atomicity and durability properties. It is also used by the Transaction Service to support the two-phase commit protocol.
- Backup and restore service: this supports the isolation property.

This view is consistent with the X/Open DTP (Distributed Transaction Processing) model which separates the transaction manager service (i.e. the implementation of a generalized two-phase commit protocol) from a resource manager that provides services for data with a life beyond process execution. This permits both transactions on transient objects and on persistent objects without transactions.

### 2.5.7 Concurrency Control Service

The Concurrency Control Service does not depend on any other service per se. Nevertheless, it is designed to work with the Transaction Service.

### 2.5.8 Query Service

The Query Service does not depend on other service but is closely related to these Object Services: Life Cycle; Persistent Object; Relationship; Concurrency Control; Transaction; Property; and Collection.

### 2.5.9 Licensing Service

The Licensing Service depends on the Event Service. It may depend on the Relationship, Property, and Query Services for some implementations. This dependency is determined by an implementation's policy definition and entry capability. The Licensing Service also depends on the Security Service, because the Licensing Service interface can use unforgeable and secure events. The Licensing Service will use Security Service interfaces to support the requirements addressed by the challenge mechanism.

### 2.5.10 Property Service

The Property Service does not depend upon other services; however, it is closely related to Collection Service.

### 2.5.11 Time Service

The Time Service does not depend upon other services.

### 2.5.12 Security Service

The Security Service does not depend upon other services.

### 2.5.13 Trader Service

The Trader Service does not depend upon other services.

### 2.5.14 Collections Service

The Collections Service does not depend upon other services; however, it is closely related to these services: Concurrency, Naming, Persistent Object, Property, and Query.

## 2.6 Relationship to CORBA

This section provides information about the relationship of other services to the CORBA specification.

### 2.6.1 ORB Interoperability Considerations: Transaction Service

Some implementations of the Transaction Service will support:

- The ability of a single application to use both object and procedural interfaces to the Transaction Service. This is described as part of the specification, particularly in the sections “The User’s View” and ‘The Implementor’s View.”
- The ability for different Transaction Service implementations to interoperate across a single ORB. This is provided as a consequence of this specification, which defines IDL interfaces for all interactions between Transaction Service implementations.
- The ability for the same Transaction Service to interoperate with another instance of itself across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)
- The ability for different Transaction Services implementations to interoperate across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)

- A critical dependency for Transaction Service interoperation across different ORBs is the handling of the `propagation_context` between ORBs. This includes the following:
  - Efficient transformation between different ORB representations of the `propagation_context`.
  - The ability to carry the ID information (typically an X/Open XID) between interoperating ORBs.
  - The ability to do interposition to ensure efficient local execution of the `is_same_transaction` operation.

### 2.6.2 Life Cycle Service

The Life Cycle Service assumes CORBA implementations support object relocation.

### 2.6.3 Naming Service

Entities that are not CORBA objects - that is to say, not objects accessed via an Object Request Broker - are used for names (in the guise of pseudo objects). In both cases the interfaces to these entities conform as closely as possible to OMG IDL while satisfying the specific service design requirements, in order to enable maximum flexibility in the future. Specifically, in the Naming Service, name objects are pseudo objects with interfaces defined in pseudo IDL (PIDL). These objects look like CORBA objects but are specifically designed to be accessed using a programming language binding. This is done for reasons based on the expected use of these objects.

### 2.6.4 Relationship Service

The Relationship Service requires CORBA Interface Repositories to support the ability to dynamically determine if an InterfaceDef conforms to another InterfaceDef, that is, if it is a subtype. This is needed to implement type constraints for particular relationships.

### 2.6.5 Persistent Object Service

The Persistent Object Service requires CORBA Interface Repositories.

### 2.6.6 General Interoperability Requirements

Interoperability between Object Services and users of Object Services implemented on different ORBs requires common RepositoryIDs be used to identify types in both systems. The types identified by these RepositoryIDs must also be consistently defined. As described in Common Object Request Broker: Architecture and Specification, Pragma Directives for Repository Id section, all CORBAservice IDL presented in this specification is implicitly preceded at file scope by the following directive:

```
#pragma prefix “omg.org”
```

Object Service Implementations that choose to extend the standard interfaces must do so by deriving new interfaces rather than by modifying the standard interfaces.

## 2.7 Relationship to Object Model

All specifications contained in this manual conform to the OMG Object Model. No additional components or profiles are required by any service.

## 2.8 Conformance to Existing Standards

In general, existing relevant standards do not have object-oriented interfaces nor are they structured in a form that is easily mapped to objects. These specifications have been influenced by existing standards, and services have been designed which minimize the difficulty of encapsulating supporting software. The naming service specification is believed to be compatible with X.500, DCE CDS and ONC NIS and NIS+.

These specifications are broadly conformant to emerging ISO/IEC/CCITT ODP standards:

- CCITT Draft Recommendations X.900, ISO/IEC 10746 Basic Reference Model for Open Distributed Computing
- ISO/IEC JTC1 SC21 WG7 N743 Working Document on Topic 9.1 - ODP Trader