Persistent Object Service Specification

Version 1.0
New edition - April 2000

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, “Need for Object Services.”

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG’s vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.

- *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

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).

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.”

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.

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.

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.

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. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

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.

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.

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.

Interface Style Consistency

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.

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.

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.

Acknowledgments

The following companies submitted parts of the *Persistent Service* specification:

- International Business Machines Corporation
- Objectivity, Inc.
- Ontos, Inc.
- Oracle Corporation
- Persistence Software
- SunSoft, Inc.
- Versant Object Technology Corporation

Service Description 1

Contents

This chapter contains the following topics.

1.1 Overview

The goal of the Persistent Object Service (POS) is to provide common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The Persistent Object Service will be used in conjunction with other object services. For example, naming, relationships, transactions, life cycle, and so forth. The Persistent Object Service has the primary responsibility for storing the persistent state of objects, with other services providing other capabilities.

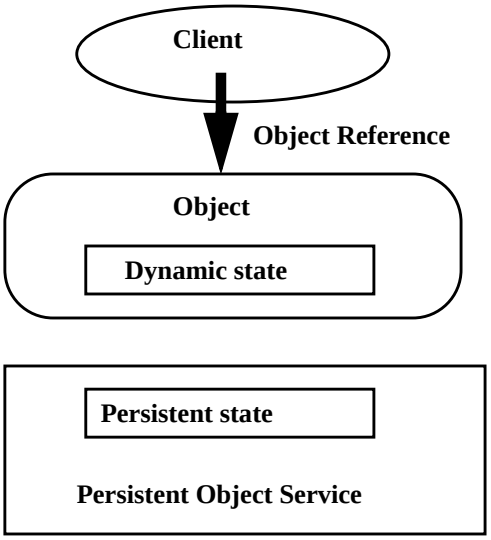

Figure 1-1 Roles in the Persistent Object Service

Figure 1-1 shows the participants in the Persistent Object Service. The state of the object can be considered in two parts: the *dynamic state*, which is typically in memory and is not likely to exist for the whole lifetime of the object (for example, it would not be preserved in the event of a system failure), and the *persistent state*, which the object could use to reconstruct the dynamic state.

Although the ORB provides the ability for an object reference to be persistent, it cannot ensure that the state of the object will be available just because the object reference is still valid.

The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. There is no requirement that any object use any particular persistence mechanism. For example, it may write its data to files using non-CORBA interfaces, or a single-level-store mechanism may be used. However, the Persistent Object Service provides capabilities that should be useful to a wide variety of objects.

Whether or not the client of an object is aware of the persistent state is a choice the object has. CORBA already provides a persistent reference handling interface (i.e., **object_to_string, string_to_object, release**). We expect that this will be sufficient for most clients to manage persistence of their referenced objects. But, because certain kinds of flexibility require the client to manage reference objects' persistence, the Persistent Object Service defines object interfaces for doing so. If this flexibility is not required, then these interfaces need not be supported or used.

The size, structure, access patterns and other properties of the dynamic and persistent state of the object varies tremendously. For many objects, their primary semantics are the efficient storage and access of its state for particular purposes. It is critical that the Persistent Object Service be able to support greatly different styles of usage and implementation in order to be useful to as many objects as possible.

As usual for object services, the primary task of this persistence specification is to define the interfaces that are needed to use the Persistent Object Service, and the conventions for how objects can work together using it.

The architecture of the Persistent Object Service defines multiple components and interfaces. In a particular situation, different parts of the service may be used. In no case does this specification assume the use of a particular implementation of a component, and it is expected that different implementations of the components will in fact work together.

1.2 Goals and Properties

The Persistent Object Service plays a key role in structuring the object system. The model of how many objects work is critically dependent on consistent and integrated use of persistence. Like other object services, the Persistent Object Service provides interfaces that can support different implementations in order to obtain different qualities of service. Those interfaces allow different components to work together.

The overall persistence architecture has multiple components. Each will be introduced in turn in this section, following presentation of some basic capabilities and properties provided by the overall architecture.

1.2.1 Basic Capabilities

The principle requirement to be supported is the need for an object to be able to make all or part of its state be persistent. Although the CORBA system defines object references as persistent (that is, they are usable until they are released regardless of the life time of their containing address space), it defined no particular way for the object to make its state persistent. The Persistent Object Service is intended ultimately to be the most common way to implement this. Therefore, there must be a way for the object to decide what state needs to be made persistent, and ways to store and retrieve that state.

It is often necessary to expose the persistent state from an object, so that the client can control the object's persistence to achieve certain types of flexibility. The Persistent Object Service defines a convention for doing this. Clients of objects sometimes need ways to refer to the persistent state, and request various operations on it. It is often not necessary to expose the persistent state from an object, so that the object implementation itself determines its persistence. In these cases, no persistence-specific object interfaces need be supported.

1.2.2 Object-oriented Storage

In existing non-object-oriented systems, persistence is accomplished by a number of data storage mechanisms. Generally, such mechanisms do not provide the key properties that object systems provide—uniform interfaces, self-description, and abstraction. The Persistent Object Service brings these properties to storage by applying object technology and principles.

1.2.2.1 Interfaces to Data

To manage object persistence, the POS defines an architecture with interfaces defined using the CORBA IDL type system. Whether detailing the particular data to be stored, describing the protocol for accessing the state, or defining the convention for making state visible for client control, the same “language” is used. This makes persistence a natural part of the software environment. These interfaces are designed to be used in a wide variety of situations, creating uniformity by encouraging most objects to support them, while allowing optimization and evolution.

By accessing data through an interface, many problems of data manipulation and exchange can be avoided. For example, programs always see data in the representation that is appropriate for the machine and programming language, of the application. Data can be translated as needed to facilitate use in different object types and implementations and for different storage formats or underlying persistent storage mechanisms (e.g., stream files, record files, or various databases) when it is accessed through the interface.

1.2.2.2 Self-description

A powerful characteristic of object-oriented systems is that the elements are self-describing. It is possible to determine from an object what kind of object it is and what interfaces it supports. In the persistence architecture this means, for example, that a client can determine whether or not an object wishes to make its persistent state visible by checking to see if the object supports the interface for doing so.

It also means that the data can be manipulated to some degree independently of the objects whose state they represent. This can allow generic facilities such as backup, migration and storage accounting, to be done independent of the objects whose state is being stored.

1.2.2.3 Abstraction

In order to support a wide and evolving set of uses, a service must be able to improve and replace its implementations without affecting the clients of that service. The desire for reuse of objects requires that those objects not depend too strictly on other objects and services, but rather be willing to work with any other components that support the required interface.

A variety of value-added products are also possible assuming that the objects depend only on the defined interfaces. By interposing unexpected implementations, for example, it may be possible to support features such as replication or versioning in a transparent way.

1.2.3 Open Architecture

A major feature of the Persistent Object Service (and the OMG architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is

particularly important for storage, where the mechanisms that are useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers may not be appropriate for mainframes.

Implementations can be lightweight, consisting of mostly library code, or powerful, leveraging decades of experience with database systems. Of course, the architecture specifies several interfaces, but also shows how new interfaces can be introduced when needed while still exploiting the rest of the architecture.

As with other object services, the Persistent Object Service is intended to be part of a collection of services. As a result, it does not attempt to solve all problems that might relate to storage. Rather, it assumes other services will provide the solutions. For example, the Persistent Object Service does not do naming, but assumes that the Name Service will perform that function; it does not do transactions, but assumes that they will be added as appropriate; it does not handle issues of general compound objects, but assumes that there will be a scheme that spans persistence, lifecycle, printing and other services.

A key idea in object systems that is critical for persistence is the ability for new and existing storage services to be able to integrate into the architecture. The requirement for such components to “plug and play” together is paramount, since one cannot expect all data to be maintained in a particular kind of file or database system. Thus, the architecture has features to allow existing databases or other storage mechanisms to be used for persistence, and for new storage mechanisms to be developed that can support both Persistent Object Service clients and other kinds of clients.

The POS architecture is open with respect to PersistentDataService, Datastore, Protocol, and PID interfaces. Although we define some minimum requirements for these in some cases, many alternatives are allowed, including ones that have not yet been defined.

1.3 Views of Service

There are multiple views of the service, and each participant may need to consider only a part of the architecture.

1.3.1 Client

It is common for clients of objects to need to control or to assist in managing persistence. In particular, the timing of when the persistent state is preserved or restored, and the identification of which persistent state is to be used for an object, are two aspects often of interest to clients. The ability of a client to see the object and its data separately allows different object implementations to be used with the same data and allows different files or databases and formats to be used with the same object implementation.

However, the client need only deal with such complexity when this type of functionality is necessary. The client of the object can be completely ignorant of the persistence mechanism, if the object chooses to hide it.

The Persistent Object Service provides an interface for objects to use when they want to expose their persistence to their clients. The interface does not completely abandon encapsulation, but gives the client visibility to those functions it needs. In fact, the client is generally unaware of how or if the object uses other parts of the Persistent Object Service.

1.3.2 Object Implementation

The object has the most involvement with the persistence, and the most options in deciding how to use it. Defining and manipulating the persistent state of the object is often the most crucial part of its implementation. The first decision the object makes is what interface to its data it needs. The Persistent Object Service captures that choice in the selection of the Protocol used by the object. Some Protocols provide simple interfaces and limited functionality, others may provide more control and more powerful operations.

The object also has the choice of delegating the management of its persistent data to other services, or maintaining fine-grained control over it. The Persistent Object Service defines a Persistent Object Manager that handles much of the complexity of establishing connections between objects and storage, allowing new components to be introduced without affecting the objects or their clients.

The object may also provide the ability for its clients to manipulate its persistent state in various ways. This is important for creating a uniform view of persistence in the system.

1.3.3 Persistent Data Service

The Persistent Data Service (PDS) actually implements the mechanism for making data persistent and manipulating it. A particular PDS supports a Protocol defining the way data is moved in and out of the object, and an interface to an underlying Datastore.

The PDS has the responsibility of translating from the object world above it to the storage world below it. It plays critical roles in identifying the storage as well as providing convenient and efficient access to it.

We define multiple kinds of PDSs, each tuned to a particular protocol and data storage mechanism, since the range of requirements for performance, cost, and qualitative features is so large. Multiple PDSs must work together to create the impression of a uniform persistence mechanism. The Persistent Object Manager provides the framework for PDSs to cooperate this way.

1.3.4 Datastore

The lowest-level interface we define is a Datastore. Although Datastore interfaces are the least visible part of the persistence architecture, it may be the most valuable, since there are so many different Datastores offering a wide spectrum of tradeoffs between availability, data integrity, resource consumption, performance and cost, and it is

expected that more will be created. By having an interface that is hidden from objects and their clients, a Datastore can provide service to any and all objects that indirectly use the Datastore interface.

The Datastore plays a key role in interoperating with other storage services. It is the manifestation in the object world of the various means of storing data that are not objects. Generally, standards for Datastore interfaces have already been defined for different kinds of data repositories—relational, object-oriented, and file systems.

Persistent Service Modules 2

Contents

This chapter contains the following sections.

2.1 Service Structure

This section presents an overview of each of the major components and how they interrelate. Subsequent sections present the OMG IDL as divided into modules, which correspond closely (but not exactly) to these components, as noted below.

The major components of the Persistent Object Service are illustrated in Figure 2-1 on page 2-3. They are:

- Persistent Identifier (PID) - This describes the location of an object's persistent data in some Datastore and generates a string identifier for that data.
- Persistent Object (PO) - This is an object whose persistence is controlled externally by its clients.
- Persistent Object Manager (POM) - This component provides a uniform interface for the implementation of an object's persistence operations. An object has a single POM to which it routes its high-level persistence operations to achieve plug and play.
- Persistent Data Service (PDS) - This component provides a uniform interface for any combination of Datastore and Protocol, and coordinates the basic persistence operations for a single object.
- Protocol - This component provides one of several ways to get data in and out of an object.
- Datastore - This component provides one of several ways to store an object's data independently of the address space containing the object.

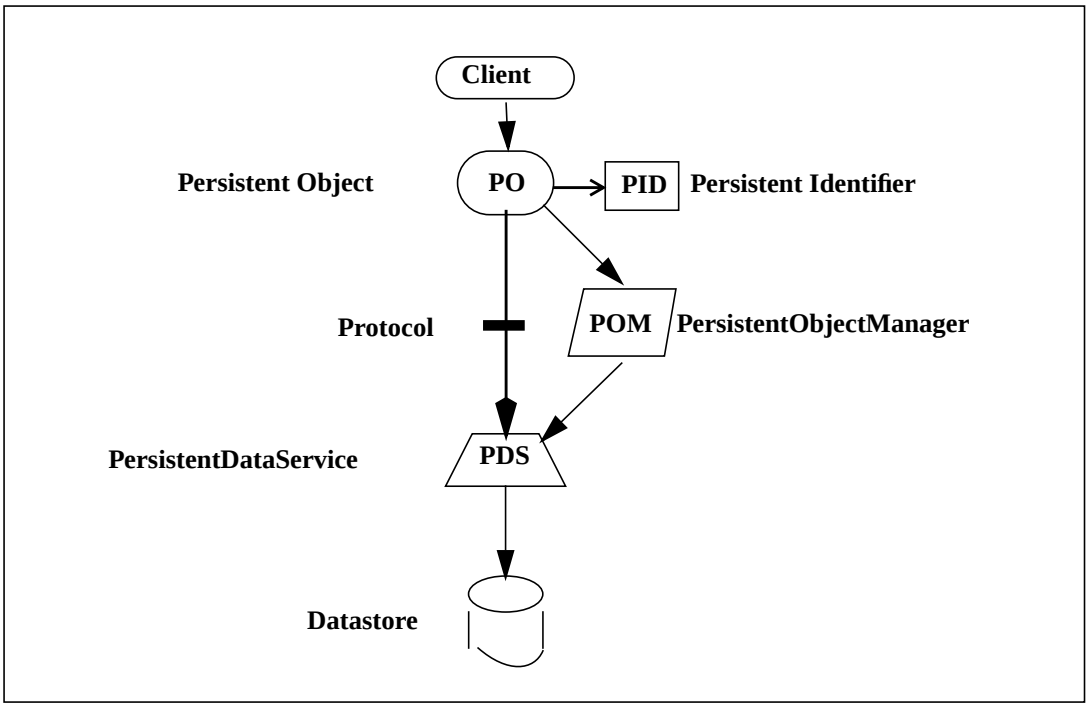

Figure 2-1 Major Components of the POS and their Interactions

The term “persistent object” is used to refer both to objects whose persistence is controlled internally or externally. Either kind of persistent object can be supported by the Persistent Object Service’s POM, PDS, Protocol and Datastore interfaces. The PO interface supports externally controlled persistence.

2.2 The CosPersistencePID Module

The **CosPersistencePID** module contains the PID Interface, the basic interface for retrieving a PID.

This section describes this interface, plus an example factory interface, and their operations in detail.

The **CosPersistencePID** Module is shown below.

```
module CosPersistencePID {

    interface PID {
        attribute string datastore_type;
        string get_PIDString();
};

};
```

The PID identifies one or more locations within a Datastore that represent the persistent data of an object and generates a string identifier for that data. An object must have a PID in order to store its data persistently. The client can create a PID, initialize its attributes, and connect it to the object. A persistent object's implementation uses the POM interface by passing the object and the PID as parameters.

The PID should not be confused with the CORBA object reference (OID). They are similar in that both have an operation that produces a string form that can be stored or communicated in whatever ways strings may be manipulated and later used to get the original PID or OID. They differ in that the PID identifies data while the OID identifies a CORBA object.

For example, assume mySpreadSheet object is referenced by both myDoc and yourDoc objects. If mySpreadSheet's OID is stored persistently with myDoc and yourDoc and then all three are brought into memory, then both documents will always see the same spreadsheet object. If mySpreadSheet's PID is stored persistently with myDoc and yourDoc and then all three objects are brought into memory, each document will see a different spreadsheet object whose states will be the same initially but will diverge over time.

2.2.1 PID Interface

The OMG IDL definition for the PID is as follows:

```
interface PID {
    attribute string datastore_type;
    string get_PIDString();
};
```

The PID contains at least one attribute:

- attribute string **datastore_type** - This identifies the interface of a Datastore. Example **datastore_types** might be "**DB2**", "**PosixFS**" and "**ObjectStore**". The PDS hides the Datastore's interface from the client, the persistent object and the POM, but PDS implementations are dependent on the Datastore's interface.

Other attributes can be added via subtyping the PID base type to reflect more specialized PIDs. Unless the **datastore_type** contains only a single object's persistent data, there is a need for more specific location information in the PID. The following example PID subtypes illustrate this:

```
#include "CosPersistencePID.idl"

interface PID_DB : CosPersistencePID::PID {
    attribute string database_name; // name of a database
};

interface PID_SQLDB : PID_DB {
    attribute string sql_statement; // SQL statement
};
```

```
interface PID_OODB : PID_DB {
    attribute string segment_name;// segment within database
    attribute unsigned long oid; //object id within a segment
};
```

The PID provides a single operation:

```
string get_PIDString();
```

This operation returns a string version of the PID called the **PIDString**. A client should only obtain the **PIDString** using the **get_PIDString** operation. This allows the PID implementation to decide the form of the **PIDString**.

Some implementations may simply concatenate the PID attributes. Others may return a more compact form specialized for specific Datastores or even databases within a Datastore. Still others may return a universally unique identifier (UUID) that facilitates movement of its persistent data either within a single Datastore or between Datastores. A UUID-based PID might be implemented by overriding the get and set attribute operations and the **get_PIDString** operation to bind and lookup the mapping between UUID and location information in a special context in the Name Service. Using such a UUID-based PID, when an object is moved, the new location would be changed by setting the attributes to indicate the new location, and the PID would make the modification in the Name Service. The **PIDString** would contain the UUID that does not change when an object's data is moved, so that references remain intact.

Some applications need to be able to restore an object given a PID but without knowing which type or implementation to use. The PID can be subtyped to accommodate this by adding the type or implementation as a PID attribute.

2.2.2 Example PIDFactory Interface

The OMG IDL definition for an example **PIDFactory** is as follows (others are also possible):

```
interface PIDFactory {
    CosPersistencePID::PID create_PID_from_key(in string key);
    CosPersistencePID::PID create_PID_from_string(
        in string pid_string);
    CosPersistencePID::PID create_PID_from_string_and_key(
        in string pid_string, in string key);
};
```

This example **PIDFactory** provides three ways of creating a PID:

```
CosPersistencePID::PID create_PID_from_key(in string key);
```

This creates an instance of a PID given a key that identifies a particular PID implementation.

CosPersistencePID::PID create_PID_from_string(in string pid_string);

This creates an instance of a PID given a **PIDString**. The **PIDString** must include some way to identify a particular PID implementation (the PID's key) in some way that allows this operation to extract the PID's key from the **PIDString**. This key identifies the PID implementation for the newly created PID.

CosPersistencePID::PID create_PID_from_string_and_key(in string pid_string, in string key);

This creates an instance of a PID whose implementation is identified by the key in the input parameter instead of the key in the **PIDString**, and whose value is determined by the **PIDString**. This is useful for when persistent data is moved between Datastores that require different PID interfaces.

2.3 The CosPersistencePO Module

The **CosPersistencePO** Module collects the interfaces that are borne by a persistent object to allow its clients and the POM to control the PO's relationship with its persistent data. This module includes two interfaces:

- The PO Interface
- The SD Interface

plus an example factory interface.

The PO interface is borne by the PO and used by the client. The SD interface is borne by the PO and used by the POM.

This section describes these interfaces and their operations in detail.

The **CosPersistencePO Module** is shown below:

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePO {

    interface PO {
        attribute CosPersistencePID::PID p;
        CosPersistencePDS::PDS connect (
            in CosPersistencePID::PID p);
    void disconnect (in CosPersistencePID::PID p);
    void store (in CosPersistencePID::PID p);
    void restore (in CosPersistencePID::PID p);
    void delete (in CosPersistencePID::PID p);
    };

    interface SD {
        void pre_store();
        void post_restore();
```

```
    };
};
```

2.3.1 The PO Interface

The PO interface provides two mechanisms for allowing a client to externally control the PO's relationship with its persistent data:

- **Connection**: This mechanism establishes a close relationship between the PO and its Datastore where the two data representations can be viewed as one for the duration of the connection. When the connection is ended, the data is the same in the PO and the Datastore, and the relationship between them no longer exists. An object can have only one connection at a time.
- **Store/restore**: These operations allow the client to move data between the PO and its Datastore in each direction separately, with each movement in each direction explicitly initiated by the client.

The PO interface operations allow client control of a single PO's persistent data. When one of these operations is performed on a PO, what data is included in these operations is up to that PO's implementation. For example, only part of the PO's private data may be included. Other POs may be included based on any criteria. If other POs are included, the target PO's implementation becomes their client and is responsible for controlling their persistence.

A PO client is responsible for the following:

- Creating a PID for the PO and initializing the PID. For storage, whatever location information is not specified will be determined by the Datastore. For a retrieval or delete operation, the location information must be complete.
- Controlling the relationship between the data in the PO and the Datastore. This is done by asking the PO to **connect()**, **disconnect()**, **store()**, **restore()**, or **delete()** itself.

The OMG IDL definition for a PO is as follows:

```
interface PO {
    attribute CosPersistencePID::PID p;
    CosPersistencePDS::PDS connect (
        in CosPersistencePID::PID p);
    void disconnect (in CosPersistencePID::PID p);
    void store (in CosPersistencePID::PID p);
    void restore (in CosPersistencePID::PID p);
    void delete (in CosPersistencePID::PID p);
};
```

The PO interface has the following operations:

CosPersistencePDS::PDS connect (in CosPersistencePID::PID p);

This begins a connection between the data in the PO and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that handles persistence for use by those Protocols that require the PO to call the PDS.

void disconnect (in CosPersistencePID::PID p);

This ends a connection between the data in the PO and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.

void store (in CosPersistencePID::PID p);

This copies the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.

void restore (in CosPersistencePID::PID p);

This copies the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.

void delete (**in CosPersistencePID::PID p**)**;**

This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

To adhere to the plug and play philosophy, objects pass these requests through to the POM, so that the interface for PO parallels that of the POM. This delegation to the POM allows objects to change PDSs (combination of Datastore and Protocol) without changing their implementation.

2.3.2 The POFactory Interface

The OMG IDL definition for an example POFactory is as follows (others are also possible):

```
#include "CosPersistencePO.idl"
// CosPersistencePO.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl

interface POFactory {
    CosPersistencePO::PO create_PO (
        in CosPersistencePID::PID p,
        in string pom_id);
};
```

The example **POFactory** provides the following operation:

CosPersistencePO::PO create_PO(in CosPersistencePID::PID p, in string pom_id);

This creates an instance of a PO that knows which POM to use and with its pid attribute already assigned.

2.3.3 The SD Interface

Some objects may be implemented knowing they are going to be persistent. Many such objects have both transient and persistent data. The Synchronized Data (SD) Interface is provided to allow such objects to synchronize their transient and persistent data. Operations on the SD are invoked only by the POM. Persistent objects whose persistence is controlled either internally or externally (PO) can support the SD interface.

The OMG IDL definition for SD is as follows:

```
interface SD {
    void pre_store();
    void post_restore();
};
```

The interface for SD provides two operations:

void pre_store();

This ensures that the persistent data are synchronized with the transient data.

void post_restore();

This ensures that the transient data are synchronized with the persistent data.

A word processing document provides a good example of how these operations might be implemented. Suppose the document type is implemented with the following data:

- text buffer (persistent)
- attributes (persistent)
- text cache (transient)
- cursor location (transient)

The document could be implemented such that all work is done in the text cache. Then at store time, the text buffer needs to be updated, since it contains the actual data that will be stored. As such, the **pre_store** operation should be implemented such that any updates in the text cache are propagated to the text buffer. The **post_restore** operation should be implemented such that the text cache is inititialized with a state consistent with the text buffer.

2.4 The CosPersistencePOM Module

The **CosPersistencePOM** module contains the interface which is borne by the POM and used by the PO. It contains a single interface, the POM Interface.

This section describes this interface and its operations in detail. The **CosPersistencePOM** module is shown below:

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePOM {

    interface Object;
    interface POM {
        CosPersistencePDS::PDS connect (
            in Object obj,
            in CosPersistencePID::PID p);
        void disconnect (
            in Object obj,
            in CosPersistencePID::PID p);
        void store (
            in Object obj,
            in CosPersistencePID::PID p);
        void restore (
            in Object obj,
            in CosPersistencePID::PID p);
        void delete (
            in Object obj,
            in CosPersistencePID::PID p);
    };
};
```

Clients of a PO will see the operations of the **POM** interface indirectly through the **PO** interface. The implementation of a persistent object with either externally or internally controlled persistence can use the **POM** interface. The **POM** provides a uniform interface across all PDSs, so different PDSs (combination of Datastore and Protocol) can be used without changing the object's implementation.

The OMG IDL definition of the **POM** is as follows:

```
interface POM {
    CosPersistencePDS::PDS connect (
        in Object obj,
        in CosPersistencePID::PID p);
    void disconnect (
        in Object obj,
        in CosPersistencePID::PID p);
    void store (
        in Object obj,
        in CosPersistencePID::PID p);
```

```
        void restore (
            in Object obj,
            in CosPersistencePID::PID p);
        void delete (
            in Object obj,
            in CosPersistencePID::PID p);
};
```

The **POM** interface has the following operations:

CosPersistencePDS::PDS connect (in Object obj, in CosPersistencePID::PID p);

This begins a connection between data in the object and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that is assigned the object's PID for use by those Protocols that require the PO to call the PDS.

void disconnect (in Object obj, in CosPersistencePID::PID p);

This ends a connection between the data in the object and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.

void store (in Object obj, in CosPersistencePID::PID p);

This gets the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.

void restore (in Object obj, in CosPersistencePID::PID p);

This gets the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.

void delete (in Object obj, in CosPersistencePID::PID p);

This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

The major function of the POM is to route requests to a PDS that can support the combination of Protocol and Datastore needed by the persistent object. To do this, the POM must know which PDSs are available and which Protocol and Datastore combinations they support. There are several possible ways that this information can be made available to a POM:

- How a Protocol is associated with an object. One possibility is for the client to set the Protocol for that object. Another possibility is for the Protocol to be associated with the object's type or implementation.

- How a POM finds out the set of available PDSs and which Protocol (or object type) and Datastores they support. One possibility is for the POM to find the information in a configuration file or a registry. Another possibility is to provide an interface to the POM for registering the information. The best or most natural technique may depend on the environment.

Because there are multiple ways to accomplish the above and more experience is needed to better understand whether there is a best way and what that might be, a POM interface for registering this information in the POM is not specified at this time.

When the POM is asked to store an object, the following steps logically occur:

1. From the PID, the POM gets the **datastore_type** attribute.
2. Regardless of how the Protocol is associated with the object, the POM uses the combination of Protocol and **datastore_type** to determine the PDS.
3. The POM passes the store request through to the PDS.
4. The PDS gets data from the object using a Protocol and stores the data in the Datastore.

The routing function of the POM serves to shield the client from having to know the details of how actual data storage/retrieval takes place. A client can change the repository of an object by changing the PID. The change will result in routing the next store/restore request to whatever the appropriate PDS is for the new Datastore.

Figure 2-2 illustrates an example of the routing logic for the storage of myDoc in a DB2 database. This figure and the following example steps assume that, for this POM, the Protocol is associated with object type:

1. The POM is asked to perform a store on myDoc with pid1.
2. The POM finds the **datastore_type** associated with pid1 (e.g., DB2).
3. The POM finds the object type of myDoc (e.g., document).
4. The POM determines that myDoc will use a particular PDS (e.g., pds1).
5. The POM routes the store/restore to pds1.
6. The PDS gets the persistent data using protocol1 and stores the data in the DB2 Datastore at pid1.

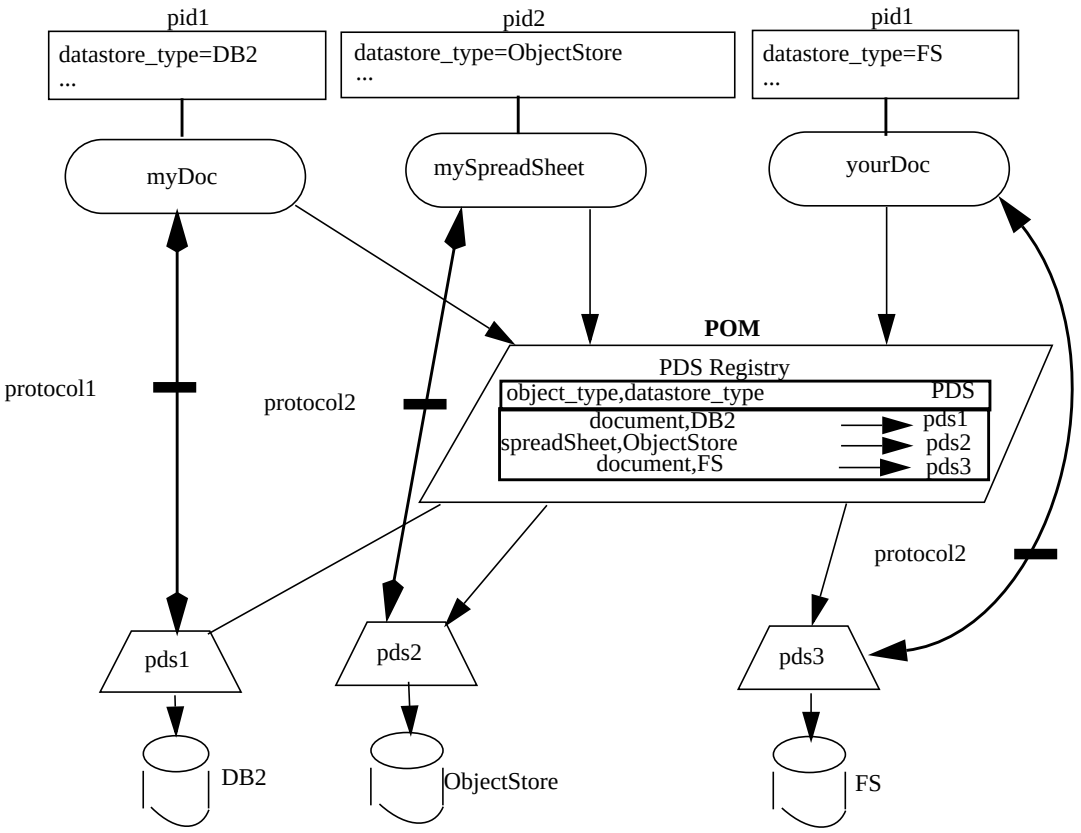

Figure 2-2 Example to illustrate POMFunctions

2.5 Persistent Data Service (PDS) Overview

The PDS implementation is responsible for the following:

- Interacting with the object to get data in and out of the object using a *protocol*. Protocols are introduced in this section; three example protocols and a discussion of additional protocols are presented in Section 2.7 through Section 2.12.
- Interacting with the Datastore to get data in and out of the object. Datastores are introduced in this section, and an example datastore plus a discussion of implementing additional datastores are presented in Section 2.13 and Section 2.14.

A PDS performs the work for moving data into and out of an object and moving data into and out of a Datastore. There can be a wide variety of implementations of PDSs which provide different performance, robustness, storage efficiency, storage format, or other characteristics, and which are tuned to the size, structure, granularity, or other properties of the object's state.

Because the range of storage requirements is so large, there may be different ways in which the object can best access its persistent data, and there may be different ways in which the PDS can store that data. The way in which the object interacts with the PDS

is called the Protocol. A Protocol may consist of calls from the object to the PDS, calls from the PDS to the object, implicit operations implemented with hidden interfaces, or some combination. The interaction might be explicit, for example, asking the object to stream out its data, or implicit, for example, the object might be mapped into persistent virtual memory. The Protocol is initiated when an object's persistent state is stored, restored, or connected; this may be initiated by a POM or by the object itself. What happens after that depends on the particular Protocol. An object that uses a particular Protocol can work with any PDS that supports that Protocol. There is no "standard" protocol. This specification defines three Protocols: the Direct Attribute (DA) Protocol, the ODMG Protocol, and the Dynamic Data Object (DDO) Protocol. A PDS might also use a programming language-specific or runtime environment-specific or other Protocol.

A PDS may use either a standard or a proprietary interface to its Datastore. A Datastore might be a file, virtual memory, some kind of database, or anything that can store information. This specification defines one Datastore interface that can be implemented by a variety of databases (Section 2.13, "Datastores: CosPersistenceDS_CLI Module," on page 2-28).

The PDS component interface is specified here as one module containing only the base PDS interface, plus one additional module per protocol. Each protocol-specific module inherits from the base module, augmenting the base functionality as needed.

2.6 The CosPersistencePDS Module

The **CosPersistencePDS** Module contains the base interface upon which protocol-specific interfaces are built. It contains a single interface: the PDS Interface.

This section describes this interface and its operations in detail.

The **CosPersistencePDS** module is shown below. Some Protocols may require specialization of the PDS interface. However, no matter what Protocol or Datastore is used, a PDS always supports at least the following interface:

```
#include "CosPersistencePID.idl"

module CosPersistencePDS {

    interface Object;
    interface PDS {
        PDS connect (in Object obj,
            in CosPersistencePID::PID p);
        void disconnect (in Object obj,
            in CosPersistencePID::PID p);
        void store (in Object obj,
            in CosPersistencePID::PID p);
        void restore (in Object obj,
            in CosPersistencePID::PID p);
        void delete (in Object obj,
            in CosPersistencePID::PID p);
```

```
    };
};
```

The exact semantics of the connect, disconnect, store, and restore operations depend on the Protocol, since there may be other steps involved in the Protocol. In all four operations, the persistent state is determined by the PID of the object.

PDS connect (in Object obj, in CosPersistencePID::PID p);

This connects the object to its persistent state, after disconnecting any previous persistent state. The persistent state may be updated as operations are performed on the object.

void disconnect (in Object obj, in CosPersistencePID::PID p);

This disconnects the object from the persistent state. It is undefined whether or not the object is usable if not connected to persistent state.

void store (in Object obj, in CosPersistencePID::PID p);

This saves the object's persistent state.

void restore (in Object obj, in CosPersistencePID::PID p);

This loads the object's persistent state. The persistent state will not be modified unless a store or other mutating operation is performed on the persistent state.

void delete (in Object obj, in CosPersistencePID::PID p);

This disconnects the object from its persistent state and deletes the object's persistent data from the Datastore location indicated by the PID.

2.7 The Direct Access (PDS_DA) Protocol

The first protocol to be described here is the **PDS_DA** or Direct Access Protocol. The Direct Access Protocol supports direct access to persistent data through typed attributes organized in data objects that are defined in a Data Definition Language (DDL). An object using this Protocol would represent its persistent data as one or more interconnected data objects. For uniformity, the persistent data of an object is described as a single data object; however, that data object might be the root of a graph of data objects interconnected by stored data object references. If an object uses multiple data objects, the object traverses the graph by following stored data object references.

An object must define the types of the data objects it uses. Those types are specified in DDL, which is a subset of the OMG Interface Definition Language (OMG IDL) in which objects consist solely of attributes. The state of the data object is accessed using the attribute access operations defined in CORBA in conjunction with the appropriate programming language mapping.

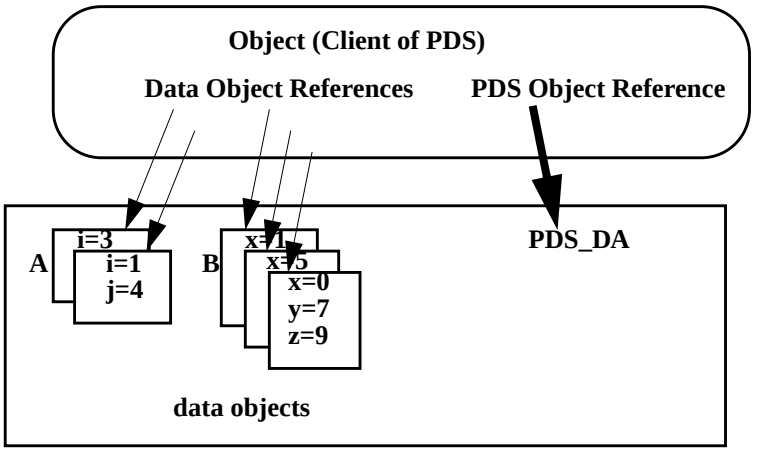

Figure 2-3 Direct Access Protocol Interfaces

The **PDS_DA** Protocol has two parts, as shown in Figure 2-3. When connected to a PDS, the object (which is effectively the client of the PDS) has an object representing the PDS which supports the **PDS_DA** interface. The object performs operations defined in the **PDS_DA** interface to get references to the data objects in the PDS. The persistent data is manipulated by performing operations using the data object references to get and set attributes on the collection of data objects in the PDS.

2.8 The CosPersistencePDS_DA Module

The **CosPersistencePDS_DA** module is a collection of interfaces which together define the protocol. This module contains the following interfaces:

- The **PID_DA** Interface
- The **DAObject** Interface
- The **DAObjectFactory** Interface
- The **DAObjectFactoryFinder** Interface
- The **PDS_DA** Interface
- The **DynamicAttributeAccess** Interface
- The **PDSClustered_DA** Interface

This section describes these interfaces and their operations in detail.

The **CosPersistencePDS_DA** module is shown below.

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePDS_DA {
```

```
    typedef string DAObjectID;

    interface PID_DA : CosPersistencePID::PID {
        attribute DAObjectID oid;
    };

    interface DAObject {
        boolean dado_same(in DAObject d);
        DAObjectID dado_oid();
        PID_DA dado_pid();
        void dado_remove();
        void dado_free();
    };

    interface DAObjectFactory {
        DAObject create();
    };

    interface DAObjectFactoryFinder {
        DAObjectFactory find_factory(in string key);
    };

    interface PDS_DA : CosPersistencePDS::PDS {
        DAObject get_data();
        void set_data(in DAObject new_data);
        DAObject lookup(in DAObjectID id);
        PID_DA get_pid();
        PID_DA get_object_pid(in DAObject dao);
        DAObjectFactoryFinder data_factories();
    };

    typedef sequence<string> AttributeNames;
    interface DynamicAttributeAccess {
        AttributeNames attribute_names();
        any attribute_get(in string name);
        void attribute_set(in string name, in any value);
    };

    typedef string ClusterID;
    typedef sequence<ClusterID> ClusterIDs;
    interface PDS_ClusteredDA : PDS_DA{
        ClusterID cluster_id();
        string cluster_kind();
        ClusterIDs clusters_of();
        PDS_ClusteredDA create_cluster(in string kind);
        PDS_ClusteredDA open_cluster(in ClusterID cluster);
        PDS_ClusteredDA copy_cluster(
            in PDS_DA source);
    };
};
```

2.8.1 The PID_DA Interface

The Persistent Identifiers (PIDs) used by the **PDS_DA** contain an object identifier that is local to the particular PDS. This value may be accessed with the following extension to the **CosPersistencePID** interface:

```
interface PID_DA : CosPersistencePID::PID {
    attribute DAObjectID oid;
};
```

The **DAObjectID** has the following attribute:

attribute DAObjectID oid();

This returns the data object identifier used by this PDS for the data object specified by the PID.The **DAObjectID** type is defined as an unbounded sequence of bytes that may be vendor-dependent.

2.8.2 The Generic DAObject Interface

The **DAObject** interface defined below provides operations that many data object clients need. A Datastore implementation may provide support for these operations automatically for its data objects. A data object is not required to support this interface. A client can obtain access to these operations by narrowing a data object reference to the **DAObject** interface:

```
interface DAObject {
    boolean dado_same(in DAObject d);
    DAObjectID dado_oid();
    PID_DA dado_pid();
    void dado_remove();
    void dado_free();
};
```

The **DAObject** has the following operations:

boolean dado_same(in DAObject d);

This returns true if the target data object and the parameter data object are the same data object. This operation can be used to test data object references for identity.

DataObjectID dado_oid();

This returns the object identifier for the data object. The scope of data object identifiers is implementation-specific, but is not guaranteed to be global.

PID_DA dado_pid();

This returns a **PID_DA** for the data object.

void dado_remove();

This deletes the object from the persistent store and deletes the in-memory data object.

void dado_free();

This informs the PDS that the data object is not required for the time being, and the PDS may move it back to persistent store. The data object must be preserved and must be brought back the next time it is referenced. This operation is only a hint and is provided to improve performance and resource usage.

2.8.3 The DAObjectFactory Interface

The scheme for factories is consistent with that of the Life Cycle Service. The factory supports the following interface:

```
interface DAObjectFactory {
    DAObject create();
};
```

The **DAObjectFactory** has the following operation:

DAObjectFactory create();

creates a new data object in the PDS.

2.8.4 The DAObjectFactoryFinder Interface

This scheme for factories follows the Life Cycle Services specification. The factory finder supports the following interface:

```
interface DAObjectFactoryFinder {
    DAObjectFactory find_factory(in string key);
};
```

The **DAObjectFactoryFinder** has the following operation:

DAObjectFactoryFinder find_factory(in string key);

This finds a factory for data objects as specified by the key.

2.8.5 The PDS_DA Interface

The DA Protocol uses an extended PDS interface called **PDS_DA**:

```
interface PDS_DA : CosPersistencePDS::PDS {
    DAObject get_data();
    void set_data(in DAObject new_data);
    DAObject lookup(in DAObjectID id);
```

```
    PID_DA get_pid();
    PID_DA get_object_pid(in DAObject dao);
    DAObjectFactoryFinder data_factories();
};
```

The **PDS_DA** provides the following operations:

DAObject get_data();

This returns the single root data object of the PDS.

void set_data(in DAObject new_data);

This sets the single root data object

DAObject lookup(in DAObjectID id);

This finds a data object by object id.

PID_DA get_pid();

This constructs a PID that corresponds to the single root data object of this PDS.

PID_DA get_object_pid(in DAObject dao);

This constructs a PID that corresponds to the specified data object, which must be in this PDS.

DAObjectFactoryFinder data_factories();

This returns a factory finder. The factory finder will provide factories for the creation of new data objects within the PDS.

2.8.6 Defining and Using DA Data Objects

A PDS_DA implements data objects that have a set of attributes defined in a Data Definition Language (DDL). DDL is a subset of OMG IDL. In DDL, all interfaces consist only of attributes; that is, there are no operations. The programming interface for accessing the persistent state is the CORBA-defined attribute access operations as specified in the particular programming language mapping. A **PDS_DA** implements those accessor operations and transfers the persistent state between the Datastore and data objects as necessary.

DA data objects are used like normal CORBA objects. They are manipulated using object references, sometimes called “data object references.” Language mappings to data object interfaces are generated just like language mappings for other interfaces.

To define a DA data object (DADO), the developer decides what state must be made persistent. For example, suppose the object's persistent data consists of two values, one integer and one floating point number. The developer would define a data object interface **MyDataObject** describing this data:

```
interface MyDataObject {
    attribute short my_short;
    attribute float my_float;
};
```

The DDL definition must be compiled, installed and linked with the object implementation as necessary for the particular PDS and CORBA environment. Mechanisms similar to those for creating stubs for IDL interfaces are used to provide the callable routines and create the runtime information necessary for the PDS implementation. The precise mechanisms are not defined in this specification.

Once the object has been connected to the PDS, the factory operations described above are used to create the data object and set it as the root object in the PDS. The object gets or sets values for the attributes using the CORBA accessor operations, for example:

```
// PDS_DA Examples
// C++ code
// Include IDL compiler output from CosPersistencePDS_DA.idl
#include "CosPersistencePDS_DA.xh"
// CosPersistencePDS_DA.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl
// connect to PDS
CosPersistencePDS_DA::PDS_DA my_pds =
    pom->connect(my_object,my_PID);
// get factory finder
DAObjectFactoryFinder daoff = my_pds->data_factories();
// get factory for MyDataObject
DAObjectFactory my_factory =
    daoff->find_factory("MyDataObject");
// create an instance of MyDataObject
MyDataObjectRef my_obj = my_factory->create();
// set the object to be the root object
my_pds->set_data(my_obj);
// put persistent state in attributes
my_obj->my_short(42);
my_obj->my_float(3.14159);
// use persistent state
my_obj->my_short(my_obj->my_short()+12);
```

The DA Protocol allows developers to build simple object implementations that just read and write attribute values whenever they need to. There is no need for an object to cache persistent data in its transient store or to explicitly request it to be read or written.

Attributes can be defined using the full flexibility of the DDL type system. A particular PDS may restrict the attribute types it supports.

A data object may contain object references to other data objects and to ordinary CORBA objects. Here is an example that extends the previous example by adding a data object reference attribute and an ordinary CORBA object reference:

```
interface MyDataObject {
    attribute short my_short;
    attribute float my_float;
    attribute MyDataObject next_data;
    attribute SomeOtherObject my_object_ref;
};
```

This example allows an instance of **MyDataObject** to refer to another instance. A Datastore implementation might restrict the scope of stored data object references. For example, it might permit only references to data objects in the same Datastore.

DDL interfaces support inheritance with semantics identical to IDL. In the following example, a new type of data object is defined that has all the attributes of **MyDataObject**, plus an additional integer:

```
interface DerivedObject : MyDataObject {
    attribute short my_extra;
};
```

Like other CORBA objects, data objects support operations on object references. In particular, the **get_interface** operation, which returns an interface repository reference to the object's most derived interface, is useful for dynamically determining the type of a data object.

2.8.7 The DynamicAttributeAccess Interface

Because data objects are CORBA objects, the CORBA Dynamic Invocation Interface can be used to get and set data object attributes dynamically, using strings to identify attributes at run time. However, to simplify dynamic access to data object attributes, the **DynamicAttributeAccess** interface is defined. This interface defines operations that allow determination of the names of the attributes of a data object and getting and setting individual attribute values by name. A data object is not required to support this interface. It can be determined whether or not a data object supports these operations by narrowing a data object reference to the **DynamicAttributeAccess** interface.

```
typedef sequence<string> AttributeNames;
interface DynamicAttributeAccess {
    AttributeNames attribute_names();
    any attribute_get(in string name);
    void attribute_set(in string name, in any value);
};
```

AttributeNames attribute_names();

This returns a sequence containing the names of the object's attributes.

any attribute_get(in string name);

This returns the value of the specified attribute.

void attribute_set(in string name, in any value);

This sets the value of the named attribute to the value specified by the any parameter.

2.8.8 The PDS_ClusteredDA Interface

It is often useful to group data objects together within a PDS. Common reasons include locking, sharing, and performance. The **PDS_ClusteredDA** is an extension to the **PDS_DA**. A non-clustered **PDS_DA** is effectively a single cluster.

Each cluster is represented as a distinct instance of the **PDS_ClusteredDA** interface, although they will typically all be implemented by the same service using the same Datastore.

In addition to supporting the normal **PDS_DA** interface, a **Clustered PDS_DA** has the following interface:

```
typedef string ClusterID;
typedef sequence<ClusterID> ClusterIDs;
interface PDS_ClusteredDA : PDS_DA {
    ClusterID cluster_id();
    string cluster_kind();
    ClusterIDs clusters_of();
    PDS_ClusteredDA create_cluster(in string kind);
    PDS_ClusteredDA open_cluster(in ClusterID cluster);
    PDS_ClusteredDA copy_cluster(
        in PDS_DA source);
};
```

ClusterID cluster_id();

This returns the id of this cluster.

string cluster_kind();

This returns the kind of this cluster.

ClusterIDs clusters_of();

This returns a sequence of ClusterIDs listing all of the clusters in this Datastore.

PDS_ClusteredDA create_cluster(in string kind);

This creates a new cluster of the specified kind in this Datastore and returns a **PDS_ClusteredDA** instance to represent it.

PDS_ClusteredDA open_cluster(in ClusterID cluster);

This opens an existing cluster that has the specified ClusterID.

PDS_ClusteredDA copy_cluster(in PDS_DA source);

Creates a new cluster, loading its state from the specified cluster, which may be implemented in a different Datastore.

2.9 The ODMG-93 Protocol

A group of Object-Oriented Database Management System (ODBMS) vendors has recently endorsed and published a common ODBMS specification called ODMG-93. That specification defines an extended version of IDL for defining ODBMS object types as well as programming language interfaces for object manipulation.

The ODMG-93 Protocol is similar to the DA Protocol, in that the object accesses attributes organized as data objects. The primary difference is that the ODMG-93 Protocol uses the Object Definition Language (ODL) defined in ODMG-93 instead of DDL, and it uses the programming language mapping defined for data objects specified in ODMG-93, rather than the CORBA IDL attribute operations.

If the ODMG-93 database object inherits the **PDS_DA** interface, then the database object can be used with the rest of this specification. Objects using the ODMG-93 Protocol would manipulate persistent data using the interfaces specified in ODMG-93.

Note that in addition to using the ODMG-93 interface as another protocol, it would be straightforward to implement the DA Protocol using an ODMG-93 ODBMS as a PDS. Since the DA Protocol is a subset of the functionality in ODMG-93, in most programming languages the language mapping for the DDL attributes would be a trivial layer on the ODMG-93 mapping. Using the ODMG-93 Protocol would fully exploit the capabilities of ODMG-93; using an ODMG-93 ODBMS to implement the DA Protocol captures those objects that use DA Protocol.

2.10 The Dynamic Data Object (DDO) Protocol

The DDO is a Datastore-neutral representation of an object's persistent data. Its purpose is to contain all of the data for a single object. Figure 2-4 illustrates an example of a DDO. A DDO has a single PID, **object_type** and set of data items whose cardinality is **data_count**. Each piece of data has a **data_name**, **data_value** and a set of properties whose cardinality is **property_count**. Each property has a **property_name** and a property value.

Although any data can be stored in a DDO, the following example illustrates how it might map onto a row in a table:

- a **DDO** = a row

- **data_count** = number of rows
- **data_item** = column
- **data_name** = column name
- **data_value** = column value
- **property_count** = number of column properties
- **property_name** = e.g., type or size
- **property_value** = e.g., character or 255

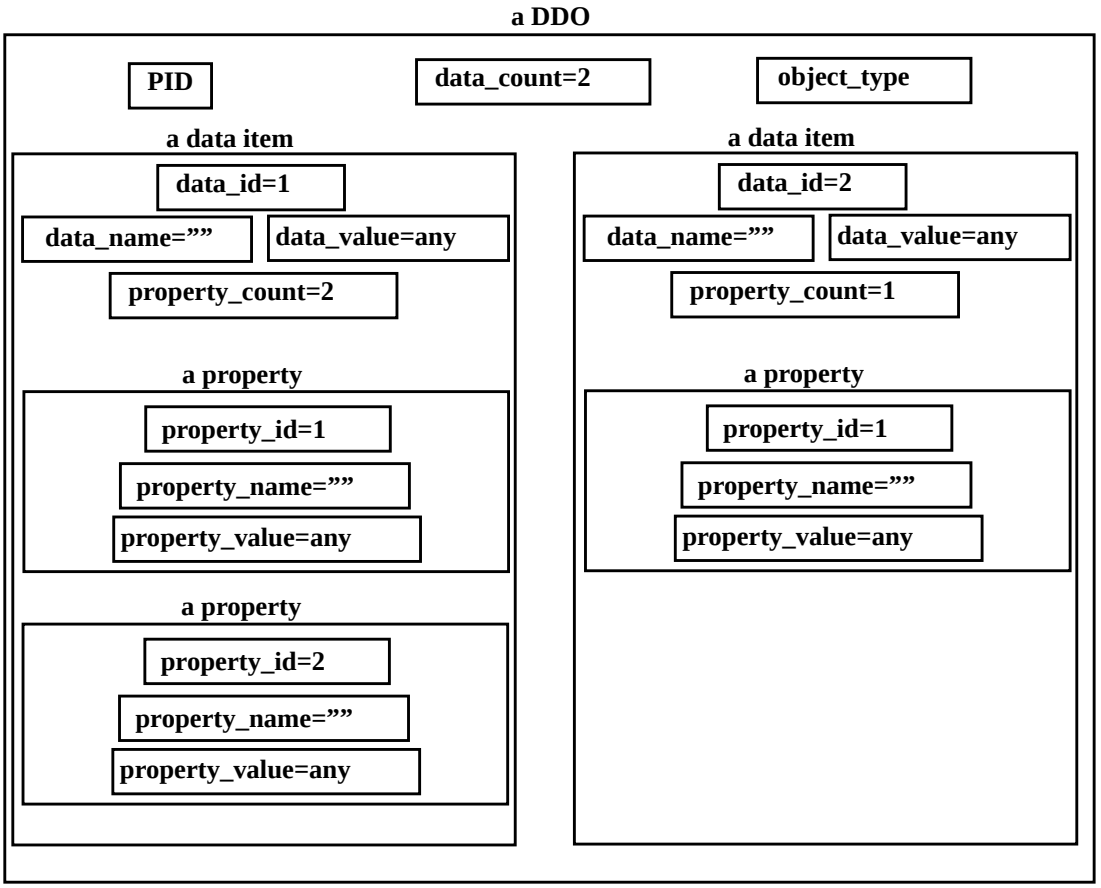

Figure 2-4 Structure of a DDO

A DDO provides a Protocol when the persistent object supports the DDO interface. In this case, the DDO interface is used to get data in and out of the persistent object. It may even provide the way that the persistent object stores its internal data, in which case a copy and reformat step is avoided.

To facilitate fast and simple storage and retrieval in specialized types of Datastore, DDOs can be used with particular conventions that are more suitable to different types of Datastore. If the DDO is used for both a Protocol and as a direct way to get data in and out of a Datastore, then copy and format costs are greatly reduced.

2.11 The CosPersistenceDDO Module

The **CosPersistenceDDO** module contains the OMG IDL to support the DDO protocol. The module contains one interface, the **DDO** interface.

This section describes the **CosPersistenceDDO** module in detail.

```
#include "CosPersistencePID.idl"

module CosPersistenceDDO {

interface DDO {
    attribute string object_type;
    attribute CosPersistencePID::PID p;
    short add_data();
    short add_data_property (in short data_id);
    short get_data_count();
    short get_data_property_count (in short data_id);
    void get_data_property (in short data_id,
        in short property_id,
        out string property_name,
        out any property_value);
    void set_data_property (in short data_id,
        in short property_id,
        in string property_name,
        in any property_value);
    void get_data (in short data_id,
        out string data_name,
        out any data_value);
    void set_data (in short data_id,
        in string data_name,
        in any data_value);
    };
};
```

A DDO has two attributes:

attribute string object_type;

This identifys the **object_type** that this DDO is associated with.

attribute CosPersistencePID::PID p;

This identify the PID of the DDO.

A DDO has the following operations for getting data in and out of the DDO:

short add_data();

This adds a new data item and returns a new **data_id** that can be used to access it.

short add_data_property (in short data_id);

This adds a new property within the data item identified by **data_id** and returns the new **property_id** that can be used to access it within the context of the data item.

short get_data_count();

This gets the number of data items in the DDO.

short get_data_property_count (in short data_id);

This gets the number of properties associated with the data item identified by **data_id**.

void get_data_property (in short data_id,
in short property_id,
out string property_name,
out any property_value);

This gets the name and value of the property identified by **property_id** within the data item identified by **data_id**.

void set_data_property (in short data_id,
in short property_id,
in string property_name,
in any property_value);

This sets the name and value of the property identified by **property_id** within the data item identified by **data_id**.

void get_data (in short data_id,
out string data_name,
out any data_value);

This gets the name and value of the data item identified by **data_id**.

void set_data (in short data_id,
in string data_name,
in any data_value);

This sets the name and value of the data item identified by **data_id**.

2.12 Other Protocols

This specification includes three protocols, but other protocols can be supported in this architecture. The proliferation of protocols would reduce the commonality of different objects, so it is desirable to use an existing protocol if that is possible. However, when

a new protocol is required, it is still possible to use other parts of the Persistent Object Service with it. In general, the protocol should be independent of the Datastore interface, although some Datastore interfaces will be better suited to some protocols.

Some protocols are already defined and are not specified here. Such standard interfaces as POSIX files are already in wide use, and there is no need to respecify them. In this case, the PID would include the file name, and the protocol would consist of reads and writes.

Other protocols are intended to be value-added and non-standard. For example, a LISP-specific PDS might take advantage of knowledge of the LISP runtime environment to create the appearance of a single-level store of LISP objects. Although such a PDS would not be usable from other programming languages, it could provide significant value to LISP programmers. Of course, it is also possible for a particular value-added protocol to be implemented as a layer on a standard Protocol.

This specification allows such protocols to be integrated in the overall POS architecture without changing that architecture.

2.13 Datastores: CosPersistenceDS_CLI Module

The last major component in the architecture is a **DataStore**, which provides operations on a data repository underneath the Protocols just discussed. As with Protocols, a variety of **DataStore** interfaces may be defined. There is no “standard” **DataStore** interface. Only one kind of **DataStore** is defined here, for record-oriented databases, because other standard interfaces already exist at this level and many customers may choose to omit this level of the architecture altogether for performance in an object-oriented database by using the DA or ODMG Protocol directly on the DBMS.

Datastore_CLI provides a uniform interface for accessing many different Datastores either individually or simultaneously. The acronym CLI refers to the X/Open Data Management Call Level Interface on which the module is based. **Datastore_CLI** is especially suited for record database and file systems (e.g., relational, IMS, hierarchical databases, and VSAM file systems) that support user sessions, connections, transactions, and scanning through data items using cursors.

The specification of this framework, where appropriate, is consistent with the X/Open CLI, IDAPI, and ODBC standards. These are industry standards which specify procedure-oriented application programming interfaces for accessing data stored in any type of Datastore.

More detailed explanations and enumeration of the options in the **Datastore_CLI** operations can be found in the X/Open CLI Specification.

DDOs are used as the way data are passed into the **Datastore_CLI** interface. If DDO is also being used as the Protocol, the PDS can use this DDO directly as a parameter to calls to the **Datastore_CLI**. When a different Protocol is being used, the PDS must create a new DO and populate it with data prior to calling the **Datastore_CLI**.

The **CosPersistenceDS_CLI** module contains the interfaces derived from ODBC and IDAPI, providing cursors into relational and other databases. The module contains the following interfaces:

- The **UserEnvironment** Interface
- The **Connection** Interface
- The **ConnectionFactory** Interface
- The **Cursor** Interface
- The **CursorFactory** Interface
- The **PID_CLI** Interface
- The **Datastore_CLI** Interface

This section describes these interfaces and their operations in detail.

The **CosPersistenceDS_CLI** Module is shown below:

```
#include "CosPersistenceDDO.idl"
// CosPersistenceDDO.idl #includes CosPersistencePID.idl

module CosPersistenceDS_CLI {
    interface UserEnvironment {
        void set_option (in long option,in any value);
        void get_option (in long option,out any value);
        void release();
    };

    interface Connection {
        void set_option (in long option,in any value);
        void get_option (in long option,out any value);
    };

    interface ConnectionFactory {
        Connection create_object (
            in UserEnvironment user_envir);
    };

    interface Cursor {
        void set_position (in long position,in any value);
        CosPersistenceDDO::DDO fetch_object();
    };

    interface CursorFactory {
        Cursor create_object (
            in Connection connection);
    };

    interface PID_CLI : CosPersistencePID::PID {
        attribute string datastore_id;
```

```
        attribute string id;
    };

    interface Datastore_CLI {
        void connect (in Connection connection,
            in string datastore_id,
            in string user_name,
            in string authentication);
        void disconnect (in Connection connection);
        Connection get_connection (
            in string datastore_id,
            in string user_name);
        void add_object (in Connection connection,
            in CosPersistenceDDO::DDO data_obj);
        void delete_object (
            in Connection connection,
            in CosPersistenceDDO::DDO data_obj);
        void update_object (
            in Connection connection,
            in CosPersistenceDDO::DDO data_obj);
        void retrieve_object(
            in Connection connection,
            in CosPersistenceDDO::DDO data_obj);
        Cursor select_object(
            in Connection connection,
            in string key);
        void transact (in UserEnvironment user_envir,
            in short completion_type);
        void assign_PID (in PID_CLI p);
        void assign_PID_relative (
            in PID_CLI source_pid,
            in PID_CLI target_pid);
        boolean is_identical_PID (
            in PID_CLI pid_1,
            in PID_CLI pid_2);
        string get_object_type (in PID_CLI p);
        void register_mapping_schema (in string schema_file);
        Cursor execute (in Connection connection,
            in string command);
    };
};
```

2.13.1 The UserEnvironment Interface

The **UserEnvironment** OMG IDL is as follows:

```
interface UserEnvironment {
    void set_option (in long option,in any value);
    void get_option (in long option,out any value);
    void release();
```

```
};
```

The **UserEnvironment** has the following operations:

void set_option (in long option, in any value);

This sets the option to the desired value. The list of settable options is specified in the X/Open CLI Specification and the IDAPI Specification.

void get_option (in long option, out any value);

This gets the value of the option. The list of gettable options is the same as that for **set_option()**.

void release();

This releases all resources associated with the **UserEnvironment**.

2.13.2 The Connection Interface

The **Connection** OMG IDL is as follows:

```
interface Connection {
    void set_option (in long option,in any value);
    void get_option (in long option,out any value);
};
```

The **Connection** interface contains the following operations:

void set_option (in long option,in any value);

This sets the option to the desired value. The list of settable options is specified in the IDAPI Specification.

void get_option (in long option, out any value);

This gets the value of the option. The list of gettable options is the same as that for **set_option**.

2.13.3 The ConnectionFactory Interface

The **ConnectionFactory** OMG IDL is as follows:

```
interface ConnectionFactory {
    Connection create_object (
        in UserEnvironment user_envir);
};
```

The **ConnectionFactory** has the following operation:

```
Connection create_object (
    in UserEnvironment user_envir);
```

This creates an instance of Connection. A Connection is created within the context of a single **UserEnvironment**.

2.13.4 The Cursor Interface

The **Cursor** OMG IDL is as follows:

```
interface Cursor {
    void set_position (in long position,in any value);
    CosPersistenceDDO::DDO fetch_object();
};
```

A cursor is a movable pointer into a list of DDOs, through which a client can move about the list or fetch a DDO from the list. The Cursor has the following operations:

void set_position (in long position, in any value);

This sets the Cursor position to the desired value. The list of settable positions is specified in the IDAPI Specification.

CosPersistenceDDO::DDO fetch_object();

This fetches the next DDO from the list, based on the current position of the Cursor.

2.13.5 The CursorFactory Interface

The **CursorFactory** OMG IDL is as follows:

```
interface CursorFactory {
    Cursor create_object (
        in Connection connection);
};
```

The **CursorFactory** has the following operations:

Cursor create_object (in Connection connection);

This create an instance of Cursor. A Cursor is created within the context of a single Connection. See the X/Open CLI Specification and IDAPI Specification for more information.

2.13.6 The PID_CLI Interface

The **PID_CLI** IDL is as follows:

```
interface PID_CLI : CosPersistencePID::PID {
    attribute string datastore_id;
    attribute string id;
};
```

PID_CLI subtypes the PID base type (see Section 2.2.1, “PID Interface,” on page 2-4), adding attributes required for the **Datatstore_CLI** interface. The **PID_CLI** interface has the following attributes:

attribute string datastore_id;

This identifies the specific datastore in use. Most datastore products support multiple datastores. For a relational database, this might be the name of a particular database containing multiple tables. For a Posix file system, this might be the pathname of a file.

attribute string id;

This identifies a particular data element within a datastore. For a relational database, this might be a table name and primary key indicating a particular row in a table. For a Posix file system, this might be a logical offset within the file indicating where the data starts.

2.13.7 The Datastore_CLI Interface

The **Datastore_CLI** OMG IDL is as follows:

```
interface Datastore_CLI {
    void connect (in Connection connection,
        in string datastore_id,
        in string user_name,
        in string authentication);
    void disconnect (in Connection connection);
    Connection get_connection (
        in string datastore_id,
        in string user_name);
    void add_object (in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void delete_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void update_object (
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    void retrieve_object(
        in Connection connection,
        in CosPersistenceDDO::DDO data_obj);
    Cursor select_object(
        in Connection connection,
        in string key);
    void transact (in UserEnvironment user_envir,
        in short completion_type);
    void assign_PID (in PID_CLI p);
    void assign_PID_relative (
        in PID_CLI source_pid,
```

```
        in PID_CLI target_pid);
    boolean is_identical_PID (
        in PID_CLI pid_1,
        in PID_CLI pid_2);
    string get_object_type (in PID_CLI p);
    void register_mapping_schema (in string schema_file);
    Cursor execute (in Connection connection,
        in string command);
};
```

In general, a client goes through the following steps to store, restore or delete DDOs:

1. Create a **UserEnvironment** and set the appropriate options to their desired values.
2. Create a **Connection** and set the appropriate options to their desired values. Open a connection to the Datastore, via **connect()**.
3. To store a DDO, call **add_object()** or **update_object()**. To restore a DDO, call **retrieve_object()**. To delete a DDO, call **delete_object()**.
4. If necessary, call **transact()** to commit or abort a Datastore transaction.
5. Repeat steps 3 and 4 as necessary.
6. Close the connection to the Datastore, via **disconnect()**. Delete the corresponding Connection.
7. Delete the **UserEnvironment**.

The **Datastore_CLI** connection operations are:

```
void connect (in Connection connection,
    in string datastore_id,
    in string user_name,
    in string authentication);
```

This opens a connection to the Datastore using the Connection. A client can establish more than one connection, but only one connection can be current at a time. The connection that **connect()** establishes becomes the current connection.

```
void disconnect (in Connection connection);
```

This closes the Connection.

```
Connection get_connection (
    in string datastore_id,
    in string user_name);
```

This returns the Connection associated with the **datastore_id**.

When any of the data manipulation operations is called, a datastore transaction begins implicitly if the Connection involved is not already active. A Connection becomes active once the transaction begins and remains active until **transact()** is called.

The **Datastore_CLI** data manipulation operations are:

void add_object (in Connection connection,
in CosPersistenceDDO::DDO data_obj);

This adds the DDO to the Datastore. If necessary, get the mapping schema information for the DDO first.

void delete_object (in Connection connection,
in CosPersistenceDDO::DDO data_obj);

This deletes the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first.

void update_object (in Connection connection,
in CosPersistenceDDO::DDO data_obj);

This updates the DDO in the Datastore. If necessary, get the mapping schema information for the DDO first.

void retrieve_object (in Connection connection,
in CosPersistenceDDO::DDO data_obj);

This retrieves the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first. To improve performance, the **DBDatastore_CLI** may obtain access to more than one DDO at a time and cache these.

Cursor select_object (in Connection connection,
in string key);

This selects and retrieve the DDO(s) which match the key from the Datastore. The DDO(s) are returned through the Cursor. If necessary, get the mapping schema information for the key first. This operation is provided to support the Query Service. In addition, the **Datastore_CLI** will support any other operation required by the Object Query Service.

The **Datastore_CLI** functions as a resource manager for the DDOs that it manages. As such, it will support all resource manager operations specified by the Transaction Service. When the Transaction Service is not being used, a transaction is initiated implicitly by either a Connection or a **transact()**, and ended with a **transact()**:

void transact (in UserEnvironment user_envir,
in short completion_type);

This completes (commit or rollback) a Datastore transaction. Transaction completion enacts or undoes any **add_object()**, **update_object()** or **delete_object()** operations performed on any Connection within the UserEnvironment since the connection was established or since a previous call to **transact()** for the same UserEnvironment. The values of **completion_type** are specified in the X/Open CLI Specification.

The **Datastore_CLI PID** Operations are:

void assign_PID (in PID_CLI p);

This assign a value for the id attribute of the pid. The first attribute, **datastore_type**, must be filled in before calling this operation. If only the first attribute is filled in, then this operation will fill in the second attribute, **datastore_id**, as well.

void assign_PID_relative (in PID_CLI source_pid,
in PID_CLI target_pid);

This assigns values for the attributes of the target_pid based on the values of the **source_pid**. The **target_pid's** first two attributes, **datastore_type** and **datastore_id**, will be assigned the same values as those of the **source_pid**. Its id attribute will be assigned a new value which is based on some relationship with that of the **source_pid**. The algorithm defining that relationship is up to the implementation.

boolean is_identical_PID (in PID_CLI pid_1, in PID_CLI pid_2);

This tests to see if the two pids are identical. In order for the two pids to be identical, the following conditions must be true:

1. Both pids must be managed by this PDS
2. all three attributes of the pids must be identical individually.

string get_object_type (in PID_CLI p);

This gets the **object_type** of the pid.

Other **Datastore_CLI** operations are:

void register_mapping_schema (in string schema_file);

This registers the mapping schema information contained within the **schema_file** with the **Datastore_CLI**. The mapping schema generally consist of individual mappings each of which is applicable to a given pair of **object_type** and **datastore_type**.

Cursor execute (in Connection connection,
in string command);

This executes a command on the Datastore. If there are any DDOs to be returned as a result, this is done through the Cursor.

2.14 Other Datastores

There are other Datastore interfaces that can be used by PDSs. Some of these interfaces are not CORBA object interfaces, in that they are not defined in IDL and the Datastores are not objects.

Some Datastores are simple, such as POSIX files. Others may be databases, and may use generic interfaces for databases and record files such as SQL, the X/Open CLI API, IDAPI or ODBC. Some Datastores are tuned to support nested documents or other specific kinds of objects such as Bento.

Because the Datastore interface is not exposed to object implementations or clients, the choice of Datastore interface is up to the PDS. So long as the PDS can support its Protocol using the particular Datastore interface, any implementation of the Datastore can be used by that PDS. The identification of data within different types of Datastores is facilitated by the PID, which can be specialized to each Datastore type.

2.15 Standards Conformance

This service is specified in standard OMG IDL.

The **Datastore_CLI** portion of the Persistent Object Service is consistent with the X/Open CLI draft standard.

The ODMG-93 PDS Object Protocol incorporates the ODMG-93 specificiation.

References *A*

The X/Open CLI standard is documented in *X/Open Data Management Call Level Interface (CLI) Draft Preliminary Specification*. Reading, UK: X/Open Ltd., 1993.

The IDAPI standard is documented in *IDAPI Working Draft*. Scotts Valley, CA: Borland International, August 1993.

The term “ODBC” refers to *Microsoft Open Database Connectivity Software Development Kit, Programmer Reference*, Version 1.0. Redmond, WA: Microsoft Corp., 1992.

The term “Bento” refers to Jed Harris and Ira Rubin, *The Bento Specification, Revision 1.0d5*. Cupertino, CA: Apple Computer, Inc., July 15, 1993,

The term “ODMG-93” refers to R.G.G.Cattell, T.Atwood, J.Duhl, G.Ferran, M.Loomis, and D.Wade, *The Object Database Standard: ODMG-93*. San Mateo, CA: Morgan Kaufmann, 1993.

A