Persistent State Service Specification

ptc/2001-12-02

Incorporates all FTF resolutions

Preface

About the 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.

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: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.

- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

The OMG collects information for each specification 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*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

Object Management Group
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781 444 0404
Fax: +1-781 444 0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:
- Ardent Software Inc.
- FUJITSU LIMITED
- IBM Corporation
- INPRISE Corporation
- IONA Technologies PLC
- Objectivity Inc.
- Object Design Inc.

- Oracle Corporation
- Persistence Software Inc.
- Secant Technologies Inc.
- Sun Microsystems Inc.
- Versant Object Technology Corporation

Service Description 1

The Persistent State Service provides a service to programmers (‘you’) who develop CORBA object implementations. A client has no way to tell if the implementation of an object uses this service.

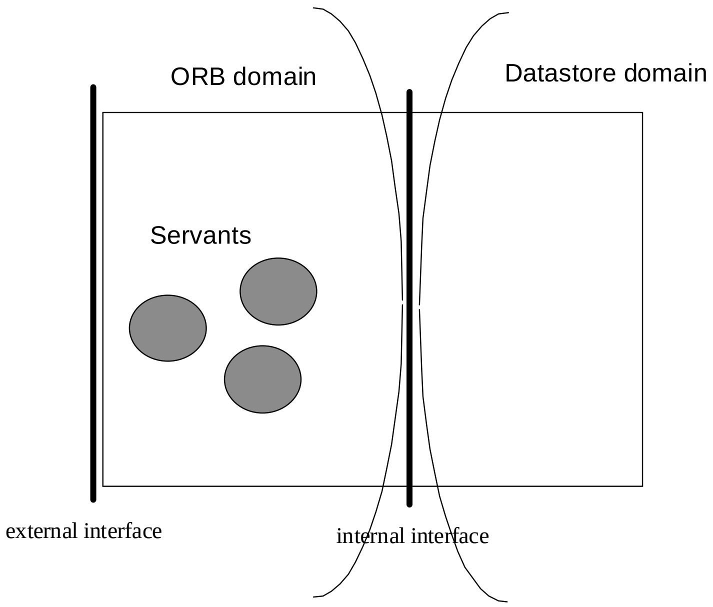

Figure 1-1 External and internal interfaces of a CORBA server

Figure 1-1 shows a computational context that hosts one or more object implementations — or CORBA server for short. This server provides an external interface that represents the interfaces supported by the object implementations it contains; these interfaces are described using IDL **interface** constructs. Servants in this server access a datastore (or several datastores) through an internal interface.

This specification focuses on the definition of this internal interface.

1.1 Fundamental Concepts

The Persistent State Service presents persistent information as storage objects stored in storage homes. Storage homes are themselves stored in datastores. A datastore is an entity that manages data, for example a database, a set of files, a schema in a relational database.

In order to manipulate a storage object, you need a programming-language object that represents it in your program. In Java and C++, this programming language object is an instance of a class: therefore we call it a storage object *instance*.

A storage object instance may be bound to a storage object in the datastore, and provides direct access to the state of this storage object: updating the instance updates the storage object in the datastore. Such a connected instance is called a storage object *incarnation*.

Likewise, to use a storage home, you need a programming language object called a storage home instance. Storage home instances themselves are provided by catalogs.

To access a storage object, you need a logical connection between your process and the datastore that contains the storage home of this storage object. This logical connection, called session, can give access to more than one datastore.

The management of sessions is either explicit (you create and manage sessions yourself) or implicit (you create one or more session pools that manage sessions for you). Sessions and session pools are the two kinds of catalogs defined by this specification.

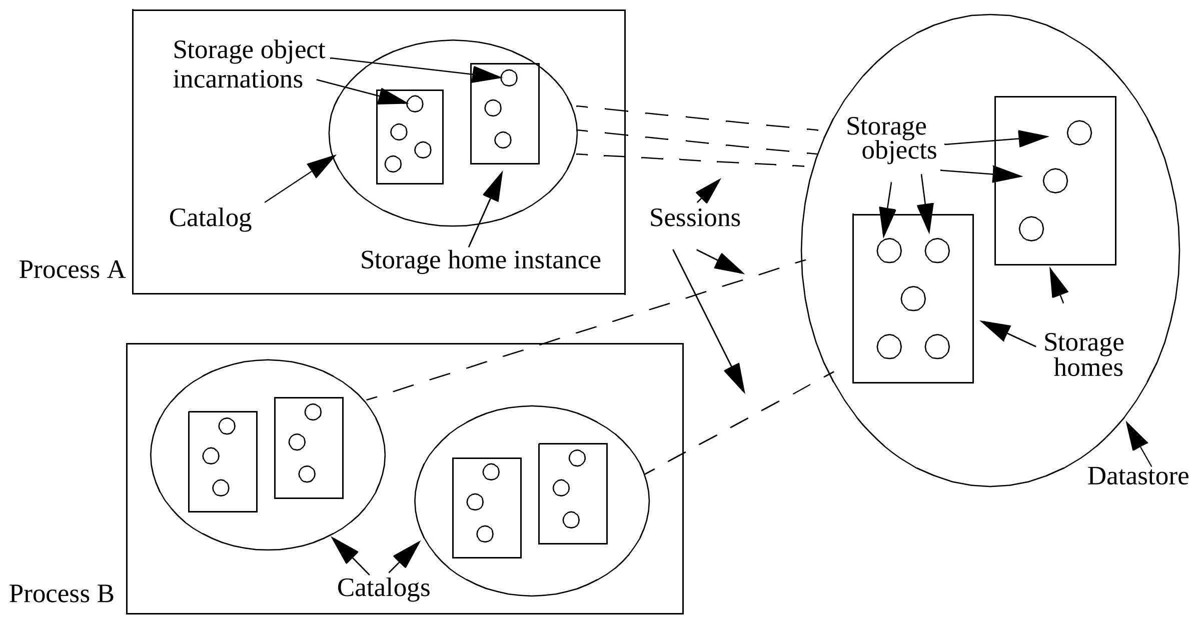

Figure 1-2 Fundamental Concepts

1.2 Datastore Model

Conceptually, a datastore is a set of storage homes. Each storage home has a type. Within a datastore, a storage home is a singleton: there is at most one storage home of a given type in this datastore.

A storage home contains storage objects. Each storage object has an ID unique within its storage home (its **short-pid**) and a global ID (its **pid**). The scope of the **pid** is all storage objects that can be accessed through the same catalog.

Each storage object has a type, which defines the state members and operations (also known as stored methods) of instances of this type. A storage object type can derive from another storage object type.

A storage home can only contain storage objects of a given type. The type of a storage home defines this storage object type, plus operations and keys (defined below). A storage home type can derive from another storage home type: the storage object type of the base storage home type must be a base of the storage object type of the derived storage home type.

Within a datastore, a storage home manages its own storage objects *and* the storage objects of all derived storage homes. A storage home and all its derived storage homes is called a storage home family.

A storage home can ensure that a list of state members of its storage object type forms a unique identifier for the storage objects it manages. Such a list of state members is called a key. A storage home can have any number of keys.

1.3 Specifying Storage Objects and Storage Homes

When developing an application with the Persistent State Service, you are responsible to specify the kind of storage objects and storage homes you need.

The Persistent State Service provides two ways to define the datastore schema and the application programming interface of storage object instances in a datastore with this schema:

- Using the Persistent State Definition Language (PSDL)
- Directly in your favorite programming language; this is known as Transparent Persistence.

PSDL is a superset of OMG IDL, with four new constructs: storagetype, storagehome, abstract storagetype and abstract storagehome.

The PSDL type model is very similar to Java: a PSDL storage type (comparable to a Java class) can implement any number of abstract storage types (comparable to Java interfaces), and can inherit from at most one other storage type. Likewise, a PSDL storage home type can implement any number of abstract storage homes and inherit from at most one other storage home type.

You provide PSDL definitions in a .psdl file. For example:

```
// In file People.psdl

abstract storagetype Person {
                    readonly state long social_security_number;
                    state string full_name;
                    state string phone_number;
};
abstract storagehome PersonHome of Person {
                    Person create(in long ssn, in string full_name, in
string phone);
};
```

A tool provided by your Persistent State Service implementation will process this file and generate code in your target programming language. For example, if your target programming language is Java, the tool will generate a Java interface for each **abstract storagetype** and **abstract storagehome**.

An **abstract storagetype** can have state members and operations.

To locate or create a storage object, you call operations on the storage home where this object is stored (or will be stored). An **abstract storagehome** can define arbitrary operations.

A PSS implementation that supports transparent persistence allows you to specify your storage objects directly in your programming language of choice. For example, you could define a JPerson Java interface as follows:

```
// Java
public interface JPerson {
   public long socialSecurityNumber();
   public String fullName();
   public void fullName(String newName);
   public String phoneNumber()
   public void phoneNumber(String newNumber);
}
```

1.4 Implementing Storage Objects and Storage Homes

A PSS implementation will typically offer several ways to define the storage types and storage home types that implement the storage object and storage home specifications you have specified in PSDL. For example, a graphical tool can let you map state members to relational columns, and abstract storagehomes to relational tables.

Storage types and storage homes can also be defined in a .psdl file, using the **storagetype** and **storagehome** constructs. A compliant PSS implementation must understand these storage type and storage home definitions and must be able to generate a full (default) implementation from these definitions alone.

For example:

```
// In file PeopleImpl.psdl

#include <People.psdl>
storagetype PersonImpl implements Person {};
storagehome PersonHomeImpl of PersonImpl implements PersonHome{};
```

A tool provided by your Persistent State Service implementation will process this file and generate code in your target programming language. For example, if your target programming language is Java, this tool will generate concrete Java classes for both PersonImpl and PersonStoreImpl.

With transparent persistence, you can define storage object implementations directly with regular programming language constructs. For example:

```
// Java
public class JPersonImpl implements JPerson {
   private long _ssn;
   private String _name;
   private String _phoneNumber;
   public long socialSecurityNumber() { return _ssn }
   // etc.
}
```

With transparent persistence, however, you cannot define application-specific storage homes: default storage homes with no keys and no operations are implicitly defined. The type hierarchy of these default storage homes parallels the type hierarchy of the corresponding storage objects. For example JPersonImpl's associated storage home type derives from java.lang.Object's associated storage home type. As a result, with transparent persistence, you can only define a single storage home family in each datastore.

1.5 Creating Sessions and Session Pools

Each PSS implementation provides a local connector object that you use to create sessions and session pools. To get a reference to a connector object, use the **find_connector** operation on the connector registry. The connector registry is a per-ORB instance singleton obtained by calling **resolve_initial_references**("PSS") on the ORB pseudo-interface. For example, the following Java code retrieves the default PSS connector associated with the ORB `myOrb`, creates a session, and then finds a storage home and inserts a new person in this storage home.

```
import org.omg.*;
CORBA.ORB myOrb = CORBA.ORB.init();
CosPersistentState.ConnectorRegistry connectorRegistry
   = CosPersistentState.ConnectorRegistryHelper.narrow(
         myOrb.resolve_initial_references("PSS")
      );
CosPersistentState.Connector connector
   = connectorRegistry.find_connector("");
// create session
CosPersistentState.Session mySession
   = connector.create_basic_session(
               org.omg.CosPersistentState.READ_WRITE,
               "",
               parameters
             );
// find person home
// (personHome is a storage home instance)
PersonHome personHome = (PersonHome)
   mySession.find_storage_home("PSDL:PersonHomeImpl:1.0");

// create person Joe Bloggs
Person joe = personHome.create(12345678, "Joe Bloggs",
                              "(617) 949-9000");
```

1.6 Transactions

Storage objects can be accessed in the context of transactions managed by the OMG Transaction Service.

When you manage sessions explicitly, a storage object incarnation and a transaction are linked through a transactional session:

- The storage object incarnation is managed by a storage home incarnation, which is itself managed by a transactional session.

- A resource object, which represents a datastore transaction, is registered with the OTS transaction.
- The transactional session is associated with the resource (datastore transaction).

A normal application developer does not tell the PSS implementation when it needs to create and register resources, or how and when it associates transactions and sessions; this is typically done by a third-party vendor, such as an implementation of the SessionPool by the PSS Vendor, or a CORBA Components container vendor.

Often, in a given CORBA server, only one resource is registered with each transaction. To retrieve the session associated with this resource, use the **current_session** operation on the connector object. For example:

```
// get the ‘current’ session
org.omg.CosPersistentState.Session mySession =
   connector.current_session();
```

In many cases, the management of sessions and their association of session with transactions is not something you want to worry about. Further, some vendors offer high performance transactional mapping and caching based on complex, highly optimized session management. When you use session pools (implicit session management), the implementation does everything for you. You have however no programmatic control over transaction/session association, and you have to use implicit transaction-context propagation. Each time you call a session pool in the context of a transaction, the session pool implementation checks if it needs to register a resource with this transaction, if it needs to create a new session, etc.

Note – Although everything is described in terms of Resources, the Persistent State Service does not provide resource objects to its users. As a result a PSS implementation does not need to use Resources to integrate with a Transaction Service implementation.

1.7 Persistent CORBA Objects

The simplest way to associate a CORBA object with a storage object is to bind the identity of the CORBA object (its **oid**, an octet sequence) with the identity of the storage object.

For example, to make the storage objects stored in storage home **PersonHomeImpl** remotely accessible, you can create for each person a CORBA object whose oid is the person’s social security number.

To make such a common association easier to implement, each storage object provides two external representations of its identity as octet sequences: the **pid** and the **short_pid**.

1.8 Relationship to CORBA Components

This specification was designed to satisfy all the requirements defined by the CORBA Components submitters. However, it does not depend on CORBA Components.

When developing a CORBA Component with container-managed persistence, a programmer sees a simplified subset of the application programming interface defined by this specification. In particular, when using container-managed persistence, a Component developer does not have access to sessions or session pools. As a result, a container vendor does not need a full Persistent State Service implementation to provide container-managed persistence.

Accessing Storage Objects 2

2.1 Introduction

Storage object instances are managed by storage home instances that are themselves managed by catalogs.

There are two kinds of catalogs: sessions and session pools. Sessions, unlike session pools, provide a programmatic control over session-allocation and session-transaction association.

Access to storage objects is also either transactional or non-transactional: this depends if you use a transactional session or transactional session pool, or not. The programming model with or without transactions is slightly different: with transactions, the application must start and end units of work (transactions). Without transactions, there is no need for demarcation.

2.2 Catalogs

A catalog is a local object that implements the local interface **CosPersistentState::CatalogBase**:

```
module CosPersistentState {

    local interface StorageHomeBase;

    exception NotFound {};

    typedef  short AccessMode;

    const AccessMode READ_ONLY  = 0;
    const AccessMode READ_WRITE =1;
    typedef CORBA::OctetSeq Pid;
    typedef CORBA::OctetSeq ShortPid;

    local interface CatalogBase {

        readonly attribute AccessMode access_mode;

        StorageHomeBase
        find_storage_home(in string storage_home_id) raises (NotFound);

        StorageObjectBase
        find_by_pid(in Pid the_pid) raises (NotFound);

        void flush();
        void refresh();
        void free_all();
        void close();
    };
};
```

The read-only attribute **access_mode** returns the access mode of this catalog. When the access mode is **READ_ONLY**, the storage object incarnations obtained through storage home instances provided by this catalog are read-only.

The **find_storage_home** operation can be used to obtain a storage home instance. **find_storage_home** raises **NotFound** if it cannot find a storage home that matches the given **storage_home_id**.

The format of the **storage_home_id** parameter is mostly implementation-defined. The **find_storage_home** operation also understands **storage_home_id** that have the form of a PSDL type id (such as "PSDL:com/acme/PersonStoreImpl:1.0", Section 3.2.4, "PSDL Type Id," on page 3-30); **find_storage_home** looks up a PSDL-defined storage home with this type id in the catalog's default datastore. If the **storage_home_id** parameter has the form ":*datastore_name*", where *datastore_name* is a string, **find_storage_home** returns a storage home instance for the storage home associated with `java.lang.Object` (Java) or `d_Object` (C++) in this datastore.

The **find_by_pid** operation attempts to locate a storage object with the given PID in the storage homes provided by the target catalog. The **find_by_pid** operation raises **NotFound** if it cannot find a storage object with this **pid**; otherwise, it returns an incarnation of this storage object.

Often, when an application creates a new storage object or updates a storage object, the modification is not written directly to disk -- the PSS implementation can cache some "dirty" data. The **flush** operation instructs the PSS implementation to write to disk any cached modifications of storage object incarnations managed by this catalog.

In addition, a PSS implementation can cache data read from the datastore(s). The **refresh** operation instructs the PSS implementation to refresh any cached storage object incarnations accessed by this catalog. This operation can invalidate any direct reference to a storage object incarnation's data member.

For example:

```
// PSDL
abstract storagetype Person {
    readonly state string full_name;
    state CORBA::OctetSeq photo;
};

// Java
Person joe = // somehow locates
             // Joe Bloggs in catalog myCatalog
byte[] photo = joe.photo();
myCatalog.refresh();
// photo is now undefined (can be an out-of-data photo,
// random octets, anything)
// joe, however, is still valid.
```

Calling **refresh** is unusual: most applications will never use this operation.

In programming languages without garbage collection, such as C++, PSDL storage object instances are reference-counted by the application. Further, when a PSDL storage object A holds a reference to another PSDL storage object B, A's instance owns a reference count of B's instance. When PSDL storage objects form a cyclic graph, the corresponding instances own reference count of each other; even if the programmer correctly releases all her reference counts, the cyclic graph will never be completely released.

For example:

```
// PSDL
abstract storagetype Person {
    readonly state string full_name;
    state ref<Person> spouse;
};
```

Once a couple is formed, each Person incarnation maintains the other Person's incarnation in memory.

The operation **free_all** deals with this issue: it instructs the catalog implementation to set the reference count of all its PSDL storage object instances to 0.

The operation **close** terminates the catalog. When closed, the catalog is also flushed. If the catalog is associated with one or more transactions (see below) when **close** is called, these transactions are marked roll-back only.

2.3 Connector

Sessions and session pools are created by connectors. A connector is a local object that represents a given PSS implementation.

Applications obtain connectors by calling the operation **resolve_initial_references** on a CORBA::ORB object.The format of the ObjectId string passed to **resolve_initial_references** is:

```
PSS[:vendor_id:implementation_id]
```

The [ ] denote optional parts in this string format.

The vendor-id is an id assigned by the OMG, and implementation-id is an implementation-defined string.

```
module CosPersistentState {

    local interface Connector;
    local interface Session;
    local interface TransactionalSession;
    local interface SessionPool;

    typedef  short TransactionPolicy;
    const TransactionPolicy NON_TRANSACTIONAL = 0;
    const TransactionPolicy TRANSACTIONAL = 1;

    struct Parameter {
        string name;
        any val;
    };

    typedef sequence<Parameter> ParameterList;
    typedef sequence<TransactionalSession> TransactionalSessionList;

    local interface Connector {
        readonly attribute string implementation_id;

        Session
        create_basic_session(
            in AccessMode access_mode,
            in ParameterList additional_parameters
        );

        TransactionalSession
        create_transactional_session(
            in AccessMode access_mode,
            in IsolationLevel default_isolation_level,
            in EndOfAssociationCallback callback,
            in ParameterList additional_parameters
        );

        TransactionalSession current_session();
```

```
            TransactionalSessionList
            sessions(
                in CosTransactions::Coordinator transaction
            );
            SessionPool
            create_session_pool(
                in AccessMode access_mode,
                in TransactionPolicy tx_policy,
                in ParameterList additional_parameters
            );

            Pid get_pid(in StorageObjectBase obj);
            ShortPid get_short_pid(in StorageObjectBase obj);

            // ...
    };
};
```

The read-only attribute **implementation_id** returns the id of this implementation.

The **create_basic_session** operation creates a basic, non-transactional, session.Typically the additional parameters will contain information such as file name, database name, or authentication information. If the implementation cannot provide a session with the desired access mode (or higher) it raises the standard exception PERSIST_STORE.

The **create_transactional_session** operation creates a new transactional session. If the implementation cannot provide the desired access mode (or higher) or the desired default isolation level, it raises the standard exception PERSIST_STORE.

The **create_session_pool** operation creates a new session pool.

The operation **sessions** returns all the transactional sessions created by this connector that are associated with resources registered with the given transaction.

Very often **sessions** will return a single session. The operation **current_session** logically calls **sessions** with the transaction associated with the calling thread; if a single session is returned, **current_session** returns it, else it raises the standard exception PERSIST_STORE.

The **get_pid** operation returns the pid of the given storage object. The **get_short_pid** operation returns the short pid of the given storage object.

2.4 Explicit Session Management

2.4.1 Overview

A PSS session is a logical connection between a process and one or more datastores. Security credentials are associated with each PSS session.

This specification defines two kinds of sessions: basic sessions for file-like access and transactional sessions for transactional access.

2.4.2 Session Local Interface

A session is a local object that supports the local interface **CosPersistentState::Session**:

```
module CosPersistentState {
    local interface Session : CatalogBase {};
};
```

2.4.3 Transactional Sessions

A transactional session is a specialized session that provides transactional access to storage objects. It supports the local interface **CosPersistentState::TransactionalSession**:

```
module CosPersistentState {

    typedef  short IsolationLevel ;
    const IsolationLevel READ_UNCOMMITTED   = 0;
    const IsolationLevel READ_COMMITTED     = 1;
    const IsolationLevel REPEATABLE_READ    = 2;
    const IsolationLevel SERIALIZABLE       = 3;


    local interface TransactionalSession : Session {

        typedef short AssociationStatus;
        const AssociationStatus NO_ASSOCIATION  = 0;
        const AssociationStatus ACTIVE          = 1;
        const AssociationStatus SUSPENDED       = 2;
        const AssociationStatus ENDING          = 3;

        readonly attribute IsolationLevel resource_isolation_level;

        void start(in CosTransactions::Coordinator transaction);
        void suspend(in CosTransactions::Coordinator transaction);
        void end(
            in CosTransactions::Coordinator transaction,
            in boolean success
        );

        AssociationStatus get_association_status();
        CosTransactions::Coordinator get_transaction();
        IsolationLevel get_isolation_level_of_associated_resource();
    };
};
```

At a given point in time, a transactional session can be associated with one resource object (a datastore transaction), or with no resource at all. The session-resource association can be active, suspended or ending. The state members of an incarnation managed by a transactional session can be used only when this session has an active association with a resource.

Typically, a resource is associated with a single session for its entire lifetime. However, with some advanced database products, the same resource may be associated with several sessions, possibly at the same time.

The **start** operation:

- re-activates a suspended (or ending) session-resource association, when the given transaction matches the transaction of the suspended (or ending) association; if there is a suspended (or ending) association but the transactions do not match, the standard exception INVALID_TRANSACTION is raised.
- else, if a resource compatible with this session is already associated with the given transaction, **start** associates this resource with this session, and makes the association active.
- else the session creates a new resource and registers it with the given transaction; it also associates itself with this resource and makes the association active.

Compatibility between resources and transactional sessions is implementation-defined. At a minimum, a resource is compatible with the session that created it.

The behavior when several resources compatible with a given session are registered with a coordinator given to **start** is implementation-defined.

The **suspend** operation suspends a session-resource association. The **suspend** operation raises the standard exception PERSIST_STORE if there is no active association, and INVALID_TRANSACTION if the given transaction does not match the transaction of the resource actively associated with this session.

The **end** operation terminates a session-resource association. The **end** operation raises the standard exception PERSIST_STORE if there is no associated resource, and INVALID_TRANSACTION if the given transaction does not match the transaction of the resource associated with this session. If the **success** parameter is **FALSE**, the resource is rolled back immediately. Like **refresh**, **end** invalidates direct references to incarnations' data members.

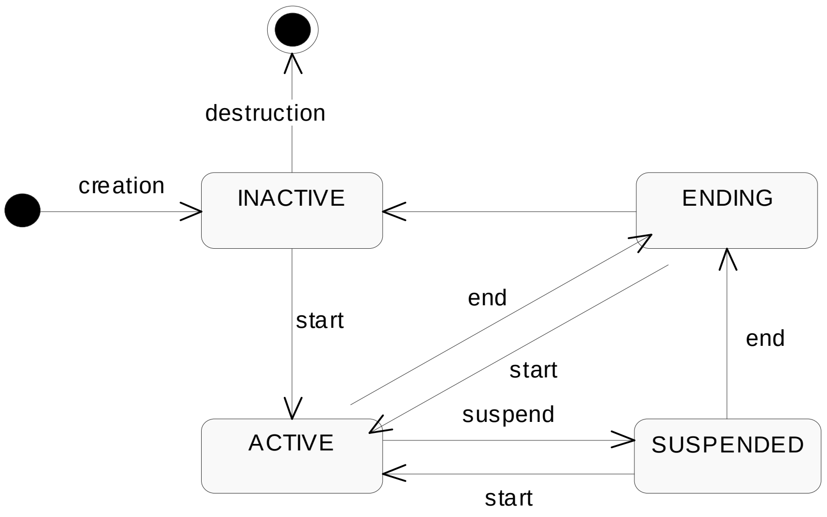

Figure 2-3 Transactional Session State Diagram

A resource can be prepared or committed in one phase only when it is not actively associated with any session. If asked to prepare or commit in one phase when still in use, the resource will rollback. A resource (provided by the PSS implementation) ends any session-resource association in which it is involved when it is prepared, committed in one phase, or rolled back.

The **get_association_status** operation returns the status of the association (if any) with this session.

The **get_transaction** operation returns the coordinator of the transaction with which the resource associated with this session is registered. **get_transaction** returns a nil object reference when the session is not associated with a resource.

When data is accessed through a transactional session actively associated with a resource, a number of undesirable phenomena may occur:

- Dirty Reads. A dirty read occurs when a resource is used to read the uncommitted state of a storage object. For example, suppose a storage object is updated using resource 1. The updated storage object’s state is read using resource 2 before resource 1 is committed. If resource 1 is rolled back, the data read with resource 2 is considered never to have existed.
- Nonrepeatable Reads. A nonrepeatable read occurs when a resource is used to read the same data twice but different data is returned by each read. For example, suppose resource 1 is used to read the state of a storage object. Resource 2 is used to update the state of this storage object and resource 2 is committed. If resource 1 is used to reread the storage object’s state, different data is returned.

Depending on the isolation level of the resource used, the application is or is not protected from these phenomena:

- when a resource has the **READ_UNCOMMITTED** isolation level, its user may experience the dirty reads and the nonrepeatable reads phenomena.
- when a resource has the **READ_COMMITTED** isolation level, its user may experience the nonrepeatable reads phenomenon, but not the dirty reads phenomenon.
- when a resource has the **SERIALIZABLE** isolation level, its user is protected from these two phenomena.

The **REPEATABLE_READ** isolation level is reserved for future use.

The **get_isolation_level_of_associated_resource** operation returns the isolation level of the resource associated with this session. If no resource is associated with this session, **get_isolation_level_of_associated_resource** raises the standard exception **PERSIST_STORE**.

The read-only attribute **resource_isolation_level** returns the isolation level of the resources created by this session.

Note that this section uses resources to describe the interaction between the Transaction Service and the Persistent State Service. The application developer, however, cannot get an object reference to such resources. This allows PSS implementations to take advantage of the non-standard direct XA integrations provided by some Transaction Service implementations.

Note – In XA terms, **start** corresponds to **`xa_start()`** with either the `TMNOFLAGS`, `TMJOIN` or `TMRESUME` flag. **end** corresponds to **`xa_end()`** with the `TMSUCCESS` or the `TMFAIL` flag. **suspend** corresponds to **`xa_end()`** with the `TMSUSPEND` or `TMSUSPEND | TMMIGRATE` flag.

2.4.4 EndOfAssociationCallback

When a session-resource association is ended, the session may not become available immediately. For example, if the session is implemented using an ODBC or JDBC connection, the PSS implementation will need this connection until the resource (ODBC/JDBC transaction) is committed or rolled back.

A session pooling mechanism may want to be notified when a session is released by the PSS implementation; this is achieved by passing a **EndOfAssociationCallback** local object to the **Connector::create_transactional_session** operation.

```
module CosPersistentState {
    local interface EndOfAssociationCallback {
        void released(in TransactionalSession session);
    };
};
```

2.5 Implicit Session Management

2.5.1 SessionPool

A session pool is a local object that implements the local interface **CosPersistentState:: SessionPool**:

```
module CosPersistentState {
    typedef sequence<Pid> PidList;

    local interface SessionPool : CatalogBase {

        void flush_by_pids(in PidList pids);
        void refresh_by_pids(in PidList pids);

        readonly attribute TransactionPolicy transaction_policy;
    };
};
```

If the transaction policy of the session pool is **NON_TRANSACTIONAL**, the **flush_by_pids** operation makes durable all of the modifications to active incarnations whose PIDs are contained in the **pids** parameter, regardless of the transactional context of the calling thread.

If the transaction policy of the target session pool is **TRANSACTIONAL**, **flush_by_pids** behaves as follows:

- If the invoking thread is associated with a transaction context, **flush_by_pids** makes durable all state modifications made in the current transactional scope for incarnations whose PIDs are contained in the **pids** parameter, flushing them to the underlying datastore.
- If the invoking thread is not associated with a transactional context, the standard exception TRANSACTION_REQUIRED is raised.

If the session pool implementation is unable to reconcile the changes and make them durable, then the PERSIST_STORE standard exception is raised.

If the current transaction policy of the session pool is **TRANSACTIONAL** and the invoking thread is associated with a transactional context, **refresh_by_pids** causes the following behavior:

- All incarnations involved in the current transaction context, and associated with the given pids, are refreshed.
- If any of the given PIDs are associated with incarnations which are themselves not associated with the current transaction, the INVALID_TRANSACTION standard exception is raised.

If the transaction policy of the session pool is **TRANSACTIONAL** and the invoking thread is not associated with a transactional context, the standard exception TRANSACTION_REQUIRED is raised.

If the session pool implementation is unable to refresh the appropriate incarnations, the PERSIST_STORE standard exception is raised.

Note – Short pids will not be passed to **flush_by_pids** and **refresh_by_pids**.

flush and **refresh**, inherited from **CatalogBase**, behave as **flush_by_pids** and **refresh_by_pids** applied to all storage object incarnations cached by the target session pool in the same context (whether transactional or not).

2.6 IThread Safety

A catalog (session or session pool) can be either thread-safe or thread-unsafe. A compliant implementation does not need to provide thread-safe catalogs.

All objects provided directly or indirectly by a thread-unsafe catalog are thread-unsafe - the application must serialize access to any of these objects, typically by using a single thread.

A storage object incarnation provided by a thread-safe catalog is like a struct: concurrent reads are safe and do not require any locking by the application; concurrent writes (or a concurrent read and a concurrent write) are not thread-safe - the application must ensure mutual exclusion to avoid problems. Flushing a storage object is like reading this object.'Refreshing' a storage object is like updating it.

Further, the following Session operations are not thread safe (for a given session): they are not supposed to be called concurrently, and no thread should be using the target session (or anything in the target session, such as an incarnation or a storage home) when they are called:

Session::free_all

Session::refresh

Session::close

TransactionalSession::start

TransactionalSession::suspend

TransactionalSession::end

OTS operations are however safe; for example one thread can call
`tx_current->rollback()`
while another thread calls start, suspend or end on a session involved in this transaction, or while a thread is using storage objects managed by that session.

Rationale:

1. Concurrent writes (or a read and a write) *within the same transaction* is extremely rare -- if PSS implementations were to provide mutual exclusion, we would penalize the common usage (single-threaded access or maybe concurrent reads) for this unusual usage.

2. Since calling these operations concurrently is wrong or at least dubious, we can avoid some locking in the PSS implementation by declaring them not thread-safe.

Defining Storage Objects 3

3.1 Introduction

The Persistent State Service provides two ways to specify datastore structures, or schemas, and the programming-language representation of storage objects and storage homes:

- with programming-language independent PSDL constructs
- directly in Java or C++

All storage object instances whether defined in PSDL or directly in Java or C++ are derived from a common base, **CosPersistentState::StorageObjectBase**. Similarly, all storage home instances implement the local interface **CosPersistentState::StorageHomeBase**:

```
module CosPersistentState {
    local interface CatalogBase;
    exception NotFound {};
    native StorageObjectBase;

    local interface StorageHomeBase {

        StorageObjectBase
        find_by_short_pid(
            in ShortPid short_pid
        ) raises (NotFound);

        CatalogBase get_catalog();
    };
};
```

StorageObjectBase maps to **`java.lang.Object`** in Java, and to **`CosPersistentState::StorageObjectBase`** in C++:

```
namespace CosPersistentState {

   class StorageObjectBase {
      protected:
         virtual ~StorageObjectBase() {}
   };
}
```

The **find_by_short_pid** operation looks for a storage object with the given short pid in the target storage home. If such an object is not found, **find_by_short_pid**, raises the CosPersistentState::NotFound exception.

The **get_catalog** operation returns the catalog that manages the target storage home instance.

3.2 PSDL Syntax and Semantics

3.2.1 Overview

Storage objects and storage object homes can be defined using the Persistent State Definition Language (PSDL).

PSDL is a superset of OMG IDL v2.4: storage objects can have state members and operations parameters of any IDL type. PSDL, like IDL, is a declarative language, not a programming language.

The mapping of PSDL constructs to several programming languages is specified in the "PSDL Language Mappings" chapter.

PSDL obeys the same lexical rules as IDL (except that it adds new keywords); its grammar is an extended IDL grammar, with new constructs to define storage objects and storage homes.

A PSDL specification can contain any IDL construct; further, local operations (on local interface, values, storage objects and storage homes) can accept parameters of PSDL types, such as a sequence of storage object references.

A source file containing PSDL constructs must have a ".psdl" extension. The file CosPersistentState.psdl contains PSDL type definitions and is implicitly included in any PSDL specification.

The description of the PSDL grammar uses the same notation as the CORBA specification:

Table 3-1 PSDL EBNF

Symbol	**Meaning**
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal

Table 3-1 PSDL EBNF *(Continued)*

Symbol	**Meaning**
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

3.2.2 Keywords

Both OMG IDL keywords and the identifiers listed in Table 3-2 are reserved for use as PSDL keywords and may not be used otherwise.

Table 3-2 PSDL Keywords

as
factory
implements
key
of
primary
ref
scope
state
storagehome
storagetype
stores
strong

PSDL also uses the IDL keywords **factory** and **const** in productions unlike the IDL productions in which they are used.

3.2.3 PSDL Grammar

The PSDL grammar is the IDL grammar plus the following productions. Productions shown in italics are defined in the CORBA specification.

```
(1)    <psdl_specification>  ::= <psdl_definition>+
(2)       <psdl_definition>  ::=  <type_dcl> ";"
                             |    <const_dcl> ";"
                             |    <except_dcl> ";"
                             |    <interface> ";"
                             |    <psdl_module> ";"
                             |    <storagehome> ";"
                             |    <abstract_storagehome> ";"
                             |    <storagetype> ";"
                             |    <abstract_storagetype> ";"
```

```
                                        |     <value> “;”
(3)                <psdl_module>   ::=  “module” <identifier>
                                        “{“ <psdl_definition>+ “}”
(4) <abstract_storagehome_name> ::= <scoped_name>
(5)      <abstract_storagetype>    ::=  <abstract_storagetype_dcl>
                                   |    <abstract_storagetype_fwd_dcl>
(6)   <abstract_storagetype_dcl>::= <abstract_storagetype_header>
                                        “{” <abstract_storagetype_body> “}”
(7)<abstract_storagetype_fwd_dcl>::= “abstract storagetype” <identifier>
(8)<abstract_storagetype_header>::=“abstract storagetype” <identifier>
                                        [ <abstract_storagetype_inh_spec>]
(9)<abstract_storagetype_body>::= <abstract_storagetype_member>*
(10)<abstract_storagetype_member>::= <psdl_state_dcl> “;”
                                   |    <storagetype_local_op_dcl> “;”
(11)<abstract_storagetype_inh_spec>::=“:” <abstract_storagetype_name>
                                        { “,” <abstract_storagetype_name> }*
(12)<abstract_storagetype_name>::=<scoped_name>
(13)              <psdl_state_dcl>   ::=  [“readonly”] “state” <psdl_state_type_spec>
                                        <simple_declarator>
                                        { “,” <simple_declarator>}*
(14)   <psdl_state_type_spec>   ::=  <base_type_spec>
                                   |    <string_type>
                                   |    <wide_string_type>
                                   |    <abstract_storagetype_ref_type>
                                   |    <scoped_name>
(15)<abstract_storagetype_ref_type>::= [“strong”] “ref” “<“
                                        <abstract_storagetype_name> “>”
(16)  <abstract_storagehome>   ::=  <abstract_storagehome_dcl>
                                   |    <abstract_storagehome_fwd_dcl>
(17)<abstract_storagehome_fwd_dcl>::=“abstract storagehome” <identifier>
(18)<abstract_storagehome_dcl>::=<abstract_storagehome_header>
                                        “{” <abstract_storagehome_body> “}”
(19)<abstract_storagehome_header>::=“abstract_storagehome” <identifier> “of”
                                        <abstract_storagetype_name>
                                        [ <abstract_storagehome_inh_spec> ]
(20)<abstract_storagehome_body>::=<storagehome_member>*
(21)<storagehome_member::=    <local_op_dcl> “;”
                                   |    <key_dcl> “;”
                                   |    <psdl_factory_dcl> “;”
(22)<abstract_storagehome_inh_spec>::=“:” <abstract_storagehome_name>
                                        { “,” <abstract_storagehome_name> }*
(23)<storagetype_local_op_dcl>::= <op_type_spec> <identifier>
                                        <parameter_dcls> [ <raises_expr> ] [“const”]
(24)               <local_op_dcl>   ::=  <op_type_spec> <identifier>
                                        <parameter_dcls> [ <raises_expr> ]
```

```
(25)                  <key_dcl>  ::=  “key” <identifier>
                                      [“(“ <simple_declarator>
                                      { “,” <simple_declarator> }*  “)”]
(26)              <storagetype>  ::=  <storagetype_dcl>
                                 |    <storagetype_fwd_dcl>
(27)          <storagetype_dcl>  ::= <storagetype_header>
                                      {“ <storagetype_body> “}”
(28)      <storagetype_fwd_dcl>  ::= “storagetype” <identifier>
(29)       <storagetype_header>  ::=  “storagetype” <identifier>
                                      [ <storagetype_inh_spec> ]
                                      [ <storagetype_impl_spec>]
                                      [<ref_rep_directive>]
(30)         <storagetype_body>  ::=  <storagetype_member>*
(31)       <storagetype_member>  ::=  <psdl_state_dcl> “;”
                                 |    <store_directive> “;”
                                 |    <storagetype_local_op_dcl> “;”
(32)    <storagetype_inh_spec>   ::=  “:” <storagetype_name>
(33)         <storagetype_name>  ::=  <scoped_name>
(34)<storagetype_impl_spec>      ::=  “implements” <abstract_storagetype_name>
                                      { “,” <abstract_storagetype_name> }*
(35)     <storagetype_ref_type>  ::=  “ref” “<“ <storagetype_name> “>”
(36)       <storagehome_scope>   ::=  “scope” <storagehome_name>
(37)          <store_directive>  ::=  “stores” <simple_declarator> “as”
                                      <psdl_concrete_state_type>
                                      [<storagehome_scope>]
(38)<psdl_concrete_state_type>::=     <storagetype_name>
                                 |    <storagetype_ref_type>
(39)        <ref_rep_directive>  ::=  “ref” “(“ <simple_declarator>
                                      { “,” <simple_declatator> }* “)”
(40)              <storagehome>  ::=  <storagehome_header>
                                      “{” <storagehome_body> “}”
(41)       <storagehome_header>  ::=  “storagehome” <identifier> “of”
                                      <storagetype_name>
                                      [ <storagehome_inh_spec> ]
                                      [ <storagehome_impl_spec>]
                                      [<primary_key_dcl>]
(42)         <storagehome_body>  ::=  <storagehome_member>*
(43)<storagehome_inh_spec>       ::=  “:” <storagehome_name>
(44)         <storagehome_name>  ::=  <scoped_name>
(45)<storagehome_impl_spec>      ::=  “implements” <abstract storagehome_name>
                                      { “,” <abstract storagehome_name> }*
(46)          <primary_key_dcl>  ::=  “primary” “key” <identifier>
                                 |     “primary” “key” “ref”
(47)         <psdl_factory_dcl>  ::=  “factory” <identifier> <factory_parameters>
(48)       <factory_parameters>  ::=  "(" <simple_declarator> [{ ","
                                      <simple_declarator> }*] ")"
                                 |    "(" ")"
```

A PSDL specification is like an IDL specification that could also contain abstract storagetype, abstract storagehome, storagetype, and storagehome definitions. The syntax is:

```
<psdl_specification>      ::=   <psdl_definition>+
<psdl_definition>:        :=    <type_dcl> “;”
                          |     <const_dcl> “;”
                          |     <except_dcl> “;”
                          |     <interface> “;”
                          |     <psdl_module> “;”
                          |     <storagehome> “;”
                          |     <abstract_storagehome> “;”
                          |     <storagetype> “;”
                          |     <abstract_storagetype> “;”
                          |     <value> “;”
<psdl_module>             ::=   “module” <identifier>
                                “{“ <psdl_definition>+ “}”
```

3.2.4 PSDL Type Id

```
module CosPersistentState {
    typedef string TypeId;
};
```

A PSDL type id is a string that identifies a PSDL type. The format of PSDL type id is the same as the IDL format of repository ids, except that the prefix is “PSDL,” not “IDL.”

The pragmas prefix and version apply to PSDL type ids in the same way as they apply to repository ids in the IDL format (see the CORBA specification).

3.2.5 Specifying Storage Objects and Storage Homes

A storage object can have both state and behavior. The visible part of its state is described by state members. Similarly, its behavior is described by operations.

For simplicity, a storage home does not have its own state, but it can have behavior. The behavior of a storage home is described by operations on its abstract or concrete storage home type(s). An abstract or concrete storagehome can also define any number of keys; each key declaration implicitly declares a pair of finder operations.

Abstract storagetypes and abstract storagehomes are abstract specifications -- like IDL interfaces. Like IDL interfaces, they support multiple inheritance, including “diamond shape” inheritance (“diamond” shape inheritance is defined in Chapter 3 of the CORBA specification).

3.2.5.1 *Abstract Storagetype*

An abstract storagetype definition satisfies the following syntax. This is almost the same syntax as an IDL interface; however, unlike an interface, an abstract storagetype cannot contain constants, or type definitions.

```
<abstract_storagetype>          ::=   <abstract_storagetype_dcl>
                                |     <abstract_storagetype_fwd_dcl>
 <abstract_storagetype_dcl>     ::=   <abstract_storagetype_header>
                                      "{" <abstract_storagetype_body> "}"
<abstract_storagetype_fwd_dcl>::=     "abstract storagetype" <identifier>
<abstract_storagetype_header>::=      "abstract storagetype" <identifier>
                                      [ <abstract_storagetype_inh_spec>]
<abstract_storagetype_body> ::=       <abstract_storagetype_member>*
<abstract_storagetype_member>::=      <abstract_storagetype_state_dcl> ";"
                                |      <storagetype_local_op_dcl> ";"
<abstract_storagetype_inh_spec>::=    ":" <abstract_storagetype_name>
                                      { "," <abstract_storagetype_name> }*
<abstract_storagetype_name>::=        <scoped_name>
```

Each **<abstract_storagetype_name>** in a **<abstract_storagetype_inh_spec>** must denote a previously *defined* abstract storagetype.

Abstract storagetype inheritance rules are like the rules for interface inheritance: an abstract storagetype may inherit from any number of base abstract storagetypes, an abstract storagetype may not be specified as a direct abstract storagetype base more than once, it is not legal to inherit two operations or state members (or an operation and an state member) with the same name, but "diamond" shape inheritance is supported. Any abstract storagetype without a base abstract storagetype, except **CosPersistentState::StorageObject**, implicitly inherits from **CosPersistentState::StorageObject**.

An abstract storagetype forward declaration declares the name of an abstract storagetype without defining it. This permits the definition of abstract storagetypes and abstract storagehomes that refer to each other. The actual definition must follow later in the PSDL specification. Multiple forward declarations of the same abstract storagetype name are legal.

3.2.5.2 *Abstract Storagetype State Members*

The abstract state of a storage object is described using state members. The syntax is:

```
<psdl_state_dcl>                ::=   ["readonly"] "state"
                                       <psdl_state_type_spec>
                                      <simple_declarator>
                                      { "," <simple_declarator>}*
```

```
<psdl_state_type_spec>       ::=   <base_type_spec>
                              |    <string_type>
                              |    <wide_string_type>
                              |    <abstract_storagetype_ref_type>
                              |    <scoped_name>
```

For each state member, a language mapping must provide a way to retrieve the state member's value and a way to set the state member's value. The optional **readonly** keyword indicates that the state member's value can only be read.

<scoped_name> must denote a previously declared [abstract or local] interface, struct, union, type, [abstract] valuetype, or a previously defined abstract storagetype.

<psdl_state_type_spec> will not be, contain or refer to a native type.

Value state members, unlike value data members of structs or valuetypes, do not support sharing semantics: when you set a state member whose type is a valuetype or an abstract valuetype, it is really a copy of the value truncated to the formal state member type which is stored.

3.2.5.3 Embedded Storage Objects and References

Like a struct can contain other structs, a storage object can contain other storage objects: such contained storage objects are said to be embedded. The lifetime of an embedded storage object is the same as the lifetime of the containing object; the embedded object does not have an identity and cannot be referenced directly.

Like a value can contain references to other values, a storage object can contain references to other storage objects. These references, like value references, support NULL and sharing semantics. The syntax for a reference to an abstract storagetype is the following:

<abstract_storagetype_ref_type>::=[“strong”] “ref” “<“ <abstract_storagetype_name> “>”

<abstract_storagetype_name> must denote a previously declared abstract storagetype.

The default value of a reference state member is NULL.

The optional **strong** keyword indicates that the referenced storage object is destroyed when the storage object holding this reference is destroyed.

3.2.5.4 Local Operations

The syntax of a local operation on a (abstract or concrete) storage home ~~or catalog~~ is:

```
<local_op_dcl>               ::=   <op_type_spec> <identifier>
                                   <parameter_dcls> [ <raises_expr> ]
```

The syntax of a local operation on a (abstract or concrete) storage type is:

<storagetype_local_op_dcl>	**::=**	**<op_type_spec> <identifier> <parameter_dcls> [<raises_expr>] [“const”]**

A “const” operation does not update any state member of the target storage object.

In a PSDL specification, each parameter of a local operation can be of a valid IDL parameter type, or of an abstract PSDL type.

3.2.5.5 *StorageObject*

The **CosPersistentState** module defines the abstract storagetype **StorageObject** as follows:

```
abstract storagetype StorageObject {
    void destroy_object();
    boolean object_exists();
    Pid get_pid();
    ShortPid get_short_pid();
    StorageHomeBase get_storage_home();
};
```

When called on an incarnation, the **destroy_object** operation destroys the associated storage object (but does *not* destroy any of its incarnation).

When called on an incarnation, the **object_exists** operation returnsTRUE if the target incarnation represents an actual storage object, FALSE if it does not.

When called on an incarnation, the **get_pid** and **get_short_pid** operations return the pid, resp. the short pid, of the associated storage object.

The standard exception PERSIST_STORE is raised when **destroy_object, get_pid** or **get_short_pid** is called on the instance of an embedded storage object.

The **get_storage_home** operation returns the storage home instance that manages the target storage object instance.

3.2.5.6 *Abstract Storagehome*

An abstract storagehome definition satisfies the following syntax:

<abstract_storagehome>:	**:=**	**<abstract_storagehome_dcl>**
	|	**<abstract_storagehome_fwd_dcl>**
<abstract_storagehome_fwd_dcl>	**::=**	**“abstract_storagehome” *<identifier>***
<abstract_storagehome_dcl>	**::=**	**<abstract_storagehome_header> “{” <abstract_storagehome_body>“}”**
<abstract_storagehome_header>	**::=**	**“abstract”“storagehome” *<identifier>* “of” <abstract_storagetype_name>**

```
                                   [ <abstract storagehome_inh_spec> ]
<abstract_storagehome_body>::=     <storagehome_member>*

<storagehome_member       ::=      <local_op_dcl> ";"
                          |        <key_dcl> ";"
                          |        <psdl_factory_dcl> ";"
```

The **<abstract_storagetype_name>** in a **<abstract_storagehome_header>** must denote a previously defined abstract storagetype.

An abstract storagehome forward declaration declares the name of an abstract storagehome without defining it. This permits the definition of abstract storagetypes and abstract storagehomes that refer to each other. The actual definition must follow later in the PSDL specification. Multiple forward declarations of the same abstract storagehome name are legal.

3.2.5.7 *Keys*

A key is a named list of one of more state members, that satisfies the following syntax:

```
<key_dcl>     ::=  "key" <identifier>
                   ["(" <simple_declarator> { "," <simple_declarator> }*  ")"]
```

"key" <identifier> is just a shortcut notation for **"key" <identifier> "("<identifier>")"**.

Each **<simple_declarator>** must be the name of a state member of the abstract storagehome's abstract storagetype (including inherited state members). For keys defined on storage homes, each **<simple_declarator>** must be the name of a state member of the storagehome's storagetype (including inherited state members).

All **<simple_declarator>** in a key declaration must be distinct.

The following types are said to be comparable:

- integral types (**octet, short**, **unsigned short**, **long, unsigned long**, **long long**, **unsigned long long**)
- fixed types
- char, wchar, string, and wstring
- sequence<octet>
- struct with only comparable members
- valuetype with only public non-valuetype comparable state members

The types of all the state members used in the definition of a key must be comparable.

A value of this list of state members uniquely identifies at most one storage object in a storage home.

With respect to language mappings, the declaration of a key *key_name* is equivalent to the declaration of the following finder operations:

```
S find_by_key_name(<parameter_list>)
        raises (CosPersistentState::NotFound);
ref<S> find_ref_by_key_name(<parameter_list>);
```

where **S** is the abstract storagehome's abstract storagetype (or concrete storagehome's storage type), and **<parameter_list>** are **in** parameters corresponding to each state member in the key declaration, in the same order. Each finder operation attempts to locate a storage object with the given key among the storage objects managed directly or indirectly by the target storage home. If a storage object with the given key is found, **find_by_key_name** returns an incarnation of this storage object, and **find_ref_by_key_name** returns a reference to this storage object. The storage home incarnation that manages the returned incarnation or reference may be the target storage home instance, or an instance of a derived storage home in the same session. If a storage object with the given key is not found, **find_by_key_name** raises the CosPersistentState::NotFound exception, and **find_ref_by_key_name** returns a NULL reference.

For example:

```
abstract storagetype Account {
     state string accno;
     state float balance;
};

abstract storagehome Bank of Account {
     key accno(accno);
     // in the language mappings, it's like:
     // Account find_by_accno(in string accno)
     //    raises (CosPersistentState::NotFound);
     // ref<Account> find_ref_by_accno(in string accno);
};
```

3.2.5.8 Factory Operations

A factory operation satisfies the following syntax:

```
<psdl_factory_dcl>::="factory" <identifier> <factory_parameters>

<factory_parameters> ::=      "(" <simple_declarator>
                              [{ "," <simple_declarator> }*]  ")"
                        |     "(" ")"
```

Each **<simple_declarator>** must be the name of a state member of the abstract storagehome's abstract storagetype (including inherited state members). For factories defined on concrete storage homes, each **<simple_declarator>** must be the name of a state member of the storagehome's storagetype (including inherited state members).

All **<simple_declarator>** in a factory declaration must be distinct.

With respect to language mappings, the definition of a factory operation *factory_name* is equivalent to the definition of the following operation:

```
S factory_name(<parameter_list>);
```

where **S** is the abstract storagehome’s abstract storagetype (resp. storagehome’s storagetype), and **<parameter_list>** are **in** parameters corresponding to each state member in the factory operation declaration, in the same order. For example:

```
abstract storagetype Account {
    state string accno;
    state float balance;
};

abstract storagehome Bank of Account {
    factory create(accno);
    // in the language mappings, it’s like:
    // Account create(in string accno) ;
};
```

3.2.5.9 Abstract Storagehome Inheritance

An abstract storagehome may inherit from any number of abstract storagehomes, with the following syntax:

```
<abstract_storagehome_inh_spec>::=“:” <abstract_storagehome_name>
                                       { “,” <abstract_storagehome_name>
}*

<abstract_storagehome_name>::=    <scoped_name>
```

Each **<abstract_storagehome_name>** in an **<abstract_storagehome_inh_spec>** must denote a previously *defined* abstract storagehome. Further, the abstract storagetype of any base abstract storagehome must be a base abstract storagetype of the abstract storagehome’s abstract storagetype, or the abstract storagehome’s abstract storagetype itself.

“diamond” shape inheritance is supported. Like IDL interfaces and PSDL abstract storagetypes, an abstract storagehome cannot inherit two operations with the same name; as a result, it cannot inherit two keys with the same name.

3.2.5.10 Sequences and Arrays

The IDL **typedef** construct can be used to define sequences and arrays of abstract storagetype, and sequences and arrays of reference to abstract storagetype. Anonymous sequences and arrays are not supported.

For example:

```
abstract storagetype Account { /* ... */};
typedef sequence<ref<Account>> AccountList;
typedef Account AccountArray[4];
```

3.2.6 Implementing Storage Objects and Storage Homes

This specification provides two constructs sufficient to define *default* implementations for storage objects and storage homes: storagetype and storagehome. Everything is implemented except operations; in particular, the PSS implementation must generate complete implementations for all state members and keys. If the stored storage type has a reference representation, only factory operations whose parameters contain all the reference representation members are generated automatically.

Of course, an implementation generated from standard PSDL definitions is unlikely to be as efficient as an implementation defined and tuned for a particular datastore.

To allow the generation of reasonably efficient default implementations for relational and relational-like datastores, PSDL storagetypes and storagehomes borrow a number of features from SQL3 user defined types and tables:

- ***Reference representation*** - Some systems, in particular many relational systems, do not have storage object identifiers, such as row ids: a storage object id is actually the value of a state member or of a list of state members. With this kind of system, it is useful to define on the storagetype itself the structure of persistent ids, rather than later in a storage home specification or definition.
- ***Scope for references*** - When a state member of a storage object is a reference to another storage object, a priori, this reference can point to a storage object stored in any storage home. PSDL, like SQL3, let you specify a scope for this reference (i.e., a storage home where the referenced storage object must be stored). This also allows some code generators to produce smaller data members for storagetypes that contain scoped references to other storagetypes.
- ***Primary key*** - A primary key is just a distinguished key, like in relational systems.
- ***Storage home inheritance*** - The rules and semantics for PSDL storage home inheritance were designed to be the same as SQL3 table inheritance rules.

3.2.6.1 Storagetype

A storagetype definition satisfies the following syntax:

```
<storagetype>              ::=    <storagetype_dcl>
                           |      <storagetype_fwd_dcl>

<storagetype_dcl>          ::=     <storagetype_header>
                                  "{" <storagetype_body> "}"

<storagetype_fwd_dcl>      ::=    "storagetype" <identifier>

<storagetype_header>       ::=    "storagetype" <identifier>
                                  [ <storagetype_inh_spec> ]
                                  [ <storagetype_impl_spec>]
                                  [<ref_rep_directive>]

<storagetype_body>         ::=    <storagetype_member>*
```

```
<storagetype_member>        ::=   <psdl_state_attr_dcl> ";"
                            |     <store_directive> ";"
                            |     <local_op_decl>;
```

A storagetype may inherit from another storagetype. The syntax is:

```
<storagetype_inh_spec>      ::=   ":" <storagetype_name>
<storagetype_name>          ::=   <scoped_name>
```

<storagetype_name> must denote a previously defined storagetype.

A storagetype may implement any number of abstract storagetypes:

```
<storagetype_impl_spec>     ::=   "implements"
                                  <abstract_storagetype_name>
                                  { "," <abstract_storagetype_name> }*
```

<abstract_storagetype_name> must denote a previously defined abstract storagetype. The same abstract storagetype cannot appear twice in a **<storagetype_impl_spec>**. However, an abstract storagetype can appear more than once in the 'implements' graph of a storagetype. For example:

```
abstract storagetype A {/* ... */ };
abstract storagetype B : A {/* ... */ };
storagetype AImpl implements A {/* ... */};
storagetype BImpl : AImpl implements B {/* ... */};
```

The first storagetype that implements an abstract storagetype in a storagetype inheritance tree is said to implement *directly* this abstract storagetype. In the example above, AImpl implements A directly; however, BImpl does not implement A directly.

3.2.6.2 Store Directive

A store directive defines how a state member is stored.

A storagetype that directly implements an abstract storagetype that declares a state member whose type is an abstract storagetype or an array or a sequence of abstract storagetypes must provide a store directive for this state member.

A storagetype that directly implements an abstract storagetype that declares a state member whose type is an abstract storagetype reference, or an array or sequence of abstract storagetype references, may provide a store directive for this state member.

The syntax is:

```
<store_directive>           ::=   "stores" <simple_declarator> "as"
                                  <psdl_concrete_attr_type>
                                  [<storagehome_scope>]
<psdl_concrete_attr_type>   ::=   <storagetype_name>
                            |     <storagetype_ref_type>
<storagehome_scope>         ::=   "scope" <storagehome_name>
```

<simple_declarator> must be the name of a state member declared in this storagetype or in one of the abstract storagetype it implements.

- If **<psdl_concrete_attr_type>** denotes a storagetype S then the type of the state member must be an abstract storagetype S’ or a sequence/array of an abstract storagetype S’, S must implement directly or indirectly S’, and **<storagehome_scope>** shall not be specified.
- If **<psdl_concrete_attr_type>** denotes a storagetype reference ref<S> then the type of the state member must be an abstract storagetype reference ref<S’> or a sequence/array of ref<S’> and S must implement directly or indirectly S’.

The storage home scope optional clause defines in which storage home referenced objects are stored. The referenced storage home must be in the same datastore as the storage object that holds this reference. **<storagehome_name>** must denote a previously defined storage home. If no **<storagehome_scope>** is specified, referenced storage objects can be stored in any storage home.

3.2.6.3 Reference Representation

A storagetype without any base storagetype can define its reference representation. The syntax is:

```
<ref_rep_directive>:        ::=     “ref” “(“ <simple_declarator>
                                    { “,” <simple_declatator> }* “)”
```

<simple_declarator> must denote a state member directly declared in this storagetype or in one of the abstract storagetypes directly implemented by this storagetype. The same **<simple_declarator>** shall not be repeated. A storagetype has at most one reference representation.

The state members that form the reference representation of a storagetype are read-only. If any of these state members is not declared read-only, the corresponding modifier and read-write accessor will always raise the standard exception PERSIST_STORE.

A reference representation also defines this list of state members as a unique identifier for the storage objects in a storage home of this storagetype.

3.2.6.4 Storagehome

A storage home definition satisfies the following syntax:

```
<storagehome>          ::=     <storagehome_header>
                       |       “{” <storagehome_body> “}”

<storagehome_header>::=        “storagehome” <identifier> “of”
                               <storagetype_name>
                               [ <storagehome_inh_spec> ]
                               [ <storagehome_impl_spec>]

<storagehome_body> ::=         <storagehome_member>*
```

```
<storagehome_member::=       <key_dcl> ";"
                    |        <local_op_dcl>;
                    |        <factory_dcl>;
```

<storagetype_name> must denote a previously defined storagetype.

A storage home may inherit from another storage home:
<storagehome_inh_spec>::=   ":" <storagehome_name>

<storagehome_name> must denote a previously defined storagehome. The storagetype of a base storagehome must be a base of **<storagetype_name>**. Further, two storagehomes in a storagehome inheritance tree cannot have the same storagetype. For example, the following specification is not legal:

```
storagetype A {/* ... */};
storagetype B : A {/* ... */};
storagehome H of A {};
storagehome H2 of B : H {};
storagehome H3 of B : H {}; // error -- B is already the storagetype
                            // of another sub-storage-home of H.
```

A storagehome may implement any number of abstract storagehomes:

```
<storagehome_impl_spec>     ::=     "implements"
                                     <abstract_storagehome_name>
                                    { "," <abstract_storagehome_name>}*
```

<abstract_storagehome_name> must denote a previously defined abstract storagehome. The same abstract storagehome cannot appear more than once in a **<storagehome_impl_spec>**. However, an abstract storagehome can appear more than once in the ‘implements’ graph of a storagehome.

The storagehome’s storagetype must implement the abstract storagetype of each of the implemented abstract storagehomes.

A storagehome is said to directly implement an abstract storagehome when it is the first storagehome in its inheritance tree to implement this abstract storagehome.

A storagehome is said to directly implement a storagetype when it is the first home to implement this storage type in its inheritance tree.

A storagehome is said to directly implement a state member when it directly implements a storagetype that contains the definition of this state member or that directly implements the abstract storagetype in which this state member is declared.

Each key declared on the abstract storagehomes implemented directly by a storagehome must use at least one state member implemented by this storagehome. For example:

```
abstract storagetype AS {
    state string name;
};
```

```
abstract storagehome ASHome {
    key name;
};

storagetype A implements AS {};
storagetype B : A {};
storagehome H of A {};
storagehome H2 of B : H implements ASHome {
    // error -- too late, name is implemented by H.
};
```

3.2.6.5 Primary Key

A storage home without any base storage home can define a key as the primary key of its storage home family:

```
<primary_key_dcl>          ::=    "primary" "key" <identifier>
                           |      "primary" "key" "ref"
```

<identifier> must denote a key declared in one of the implemented abstract storagehomes.

primary key ref tells the PSS implementation to use the state members of the reference representation as the primary key.

3.3 Transparent Persistence

3.3.1 Overview

A PSS implementation that supports the definition of storage objects directly in Java or C++ is said to support transparent persistence. With this capability, there is no need for a separate PSDL specification of the schema. The transparent persistence mechanism also attempts to allow any Java class to be made persistent, although there are a few restrictions which are mentioned in the following sections.

The most visible benefit is that state members may be directly represented with fields (or member variables) rather than requiring accessor and modifier methods that make calls to the PSS implementation. To provide this benefit, PSS implementations that provide transparent persistence need to make sure that an object's incarnation is loaded before the program tries to access a state member from it. It also needs to be able to determine which objects have changed and need to be committed. The approaches used to accomplish these tasks dictate some of the restrictions this standard makes on the classes that can be made persistent.

Because both schema definition and data manipulation are accomplished directly in Java or C++, the majority of the description of transparent persistence is in terms of each of these languages.

3.3.2 Java

In Java, there are four techniques that are likely to be used by a PSS implementation that provides transparent persistence:

- a Java pre-processor inserts Java code to fetch objects from the database before every read of persistent-capable class fields, and code to mark objects dirty before every write to these fields;
- a special Java compiler makes these same kinds of modifications;
- a post-processor makes similar modifications, but to the bytecode that is generated by the Java compiler, rather than to source;
- a special Java virtual machine uses non-standard hooks for fetching and dirtying objects when they are read or modified.

The first two approaches require that source code be available for any class that is to be made persistent-capable, the third requires that the bytecode files (e.g. ".class" files) be available. This leads to the following specification:

A PSS implementation that supports transparent persistent must be able to make any class persistent-capable that:

- it has Java source code for;
- inherits from nothing or inherits from a class that can be made persistent-capable;
- has only fields that are one of the following:
  - a primitive data type
  - a persistent-capable class
  - of type Object
  - an Array
  - one of the following immutable classes:
    String, Character, Boolean, Byte, Short, Integer, Long, Float, or Double.

Note, in particular, that there is no requirement that a class must inherit from an incarnation base class in order to be persistent-capable.

Object identity is not necessarily maintained for objects of the immutable classes listed above. So, for example, two String fields that refer to the same String object in one transaction may refer to different objects in another; or, two String fields that refer to different String objects with the same values in one transaction, may refer to the same String object in a later transaction.

IDL types can be used for fields by using the corresponding Java constructs from the CORBA IDL to Java mapping standard. References to CORBA objects can be stored in persistent objects as their string representations (use the CORBA.object_to_string method). The application then has to explicitly convert back from a string to an object reference, and should deal with the contingencies that the object is no longer available.

Static fields and transient fields may be included in persistent capable classes, although they are not made persistent.

PSS implementations need not guarantee that incarnations are correctly loaded when an incarnation's fields are accessed through the reflection API or through JNI.

The following is an example of Java classes that can be used to define the storage object types Bank and Account:

```
// Java
public class Bank {
   public String name;
}
public class Account {
   public long id;
   public Bank myBank;
   public float balance;
}
```

The fields of the Bank and Account instances are automatically fetched and stored by the PSS implementation; that is, they are transparently persistent.

3.3.2.1 Making Objects Persist

In Java, storage homes for transparent persistent objects implement the Java interface **`org.omg.CosPersistentState.JStorageHome`**:

```
package CosPersistentState;
public interface JStorageHome extends StorageHomeBase {
   public void persist(Object obj);
}
```

An instance of a persistent-capable class can be made persistent by calling the **`persist`** method on an instance of a storage home. The **`persist`** method records the association between the object and the family of this storage home.

When the transaction is committed, every field of every persistent incarnation must be either:

- a primitive value,
- null,
- a reference to a persistent incarnation associated with the same catalog as the referring object,
- or reference to an object of one of the immutable classes listed above.

If there is a reference to a transient object, the behavior is implementation-defined. Some vendors may choose to automatically migrate the referred object to become persistent in the same database as the referring object. Other vendors may choose to raise an exception.

A new transaction must acquire its first incarnation by using some known **pid** with the **find_by_pid** operation on **CatalogBase**, or with the **find_by_short_pid** operation on **StorageHomeBase** (see Section 3.2.1, “Overview,” on page 3-26). Subsequent incarnations can then be acquired through navigation, or by additional calls to **find_by_pid** (or **find_by_short_pid**).

When an incarnation is used outside of the transaction in which it was either fetched or created, either the incarnation will hold the current valid contents of the storage object, or the incarnation will have old data, but attempting to commit changes made after reading such an out-of-date object will cause the transaction to abort. Valid data for all incarnations can be guaranteed by calling **CatalogBase::refresh**.

3.3.3 C++

A PSS implementation that provides transparent persistence for C++ must implement the Object Data Management Group (ODMG) version 2.0 standard for C++ ([ODMG] chapter 5), with the following modification: **d_Object** inherits (with public virtual inheritance) from **CosPersistentState::StorageObjectBase.**

The ODMG C++ standard uses a smart pointer class (**d_Ref**) for determining which objects need to be fetched into application memory and requires an explicit call to a **mark_modified()** member function for determining which objects have changed.

ODMG defines a **d_Transaction** class for handling transaction semantics. In PSS, transactions are handled entirely through the Transaction Service; therefore, the **d_Transaction** class should not be used in conjunction with PSS.

Also [ODMG] uses the term database for datastore.

PSDL Language Mappings 4

4.1 Introduction

Application code that uses the Persistent State Service interacts with abstract storagetypes, abstract storagehomes and types defined in the **CosPersistentState** module. Such code can be completely shielded from PSS-implementation dependencies: in C++ and Java, it should not be necessary to recompile this application code when switching from one PSS implementation to another one. To make this possible, each language mapping must fully specify the mapping for abstract storagetypes, abstract storagehomes, and the types defined by the **CosPersistentState** module.

On the other hand, storagetypes and storagehomes are mapped to concrete programming language constructs with implementation-dependent parts (such as C++ members, Java fields and methods). Language mappings should avoid to put restrictions on these concrete constructs.

Of course, each PSDL language mapping should try to be as consistent as possible with the IDL mapping. In particular, the mapping for PSDL modules in a given programming language shall be the same as the mapping for IDL modules in this language. The mapping for PSDL abstract storagetypes and abstract storagehomes should be similar to the mapping for IDL structs or abstract valuetypes; the mapping for storagetypes and storagehomes should be similar to the mapping for IDL structs or valuetypes.

Implementations of PSDL operations declared on abstract storagetype and abstract storagehomes are typically provided in classes derived from classes generated by the PSDL compiler. The PSS implementation needs factories in order to create instances of such user-defined classes. Factories for storage object instances are represented by the native type **CosPersistentState::StorageObjectFactory**, factories for storage home instances are represented by the native type **CosPersistentState::StorageHomeFactory**, factories for session instances are represented by the native type **CosPersistentState::SessionFactory**, and factories for session-pools are represented by the native type

CosPersistentState::SessionPoolFactory. The connector of a PSS implementation provides an operation to register storage object factories, **register_storage_object_factory**, an operation to register storage home factories, **register_storage_factory**, an operation to register session factories, **register_session_factory**, and an operation to register session pool factories, **register_session_pool_factory:**

```
module CosPersistentState {
    native StorageObjectFactory;
    native StorageHomeFactory;
    native SessionFactory;
    native SessionPoolFactory;

    interface Connector {

        StorageObjectFactory
        register_storage_object_factory(
            in TypeId storage_type_name,
            in StorageObjectFactory factory
        );

        StorageHomeFactory
        register_storage_home_factory(
            in TypeId storage_home_type_name,
            in StorageHomeFactory factory
        );

        // ...
    };
};
```

Each **register_** operation returns the factory previously registered with the given name; they return NULL when there is no previously registered factory.

The **CosPersistentState** module also defines two enumeration types:

- **YieldRef**, which can be used to define overloaded functions or methods that return incarnations and references.
- **ForUpdate**, which can be used to define overloaded accessor function/method which will update the state member.

```
module CosPersistentState {
    enum YieldRef { YIELD_REF };
    enum ForUpdate { FOR_UPDATE };
};
```

4.2 Java Mapping

4.2.1 Abstract Storagetypes

An abstract storagetype definition is mapped to a public Java interface with the same name and the definition of the associated Holder class.

Refs are mapped to pids (byte[]) in Java.

The mapped Java interface extends the mapped interfaces of all the abstract storagetype inherited by this abstract storagetype.

For example:

```
// PSDL
abstract storagetype A {}; // implicitly inherits
                        // CosPersistentState::StorageObject
abstract storagetype B : A {};
```

is mapped to:

```
// Java
public interface A
   extends CosPersistentState.StorageObject {}

public interface B extends A {}
```

The forward declaration of an abstract storagetype is mapped to the forward declaration of its mapped interface and the associated Holder class.

4.2.2 Arrays and Sequences

Like arrays and sequences of IDL types, arrays and sequences of abstract storagetypes and reference to abstract storagetype are mapped to Java arrays. For example:

```
// PSDL
abstract storagetype A {};
typedef sequence <ref<A>> ASeq;
```

ASeq is mapped to **ARef[]**.

Holder classes are also generated, like for IDL types.

4.2.3 State Members

Each state member is mapped to a number of overloaded accessor and modifier methods, with the same name as the state member. These methods can raise any CORBA standard exception.

A state member whose mapped Java type is immutable is simply mapped to a pair of accessor and modifier methods. There is no modifier method if the state member is read-only.

For example:

```
// PSDL
abstract storagetype Person {
   state string name;
};
```

is mapped to

```
// Java
public interface Person extends StorageObject {
   public String name();
   public void name(String s);
};
```

A state member whose type is a abstract storagetype is also mapped to a pair of accessor and modifier methods, or just an accessor method when the state member is read-only.

A state member whose type is a reference to an abstract storagetype is mapped to two accessors and two modifier methods. One of the accessor methods takes no parameter and returns a storage object incarnation, the other takes a **CosPersistentState.YieldRef** parameter and returns a reference. One of the modifier methods takes an incarnation, the other one takes a reference. If the state member is read-only, only the accessor methods are generated. For example:

```
abstract storagetype Bank;
abstract storagetype Account {
   state long id;
   state ref<Bank> my_bank;
};
```

is mapped to:

```
// Java
public interface Account
    extends CosPersistentState.StorageObject {
   public long id();
   public void id(long l);
   public Bank my_bank();
   public byte[] my_bank(CosPersistentState.YieldRef yr);
   public void my_bank(Bank k);
   public void my_bank(byte[] kr);
}
```

All other state members are mapped to two accessor methods (one read-only, one read-write) and one modifier method. If such a state member is read-only, only the read-only accessor is generated. For example:

```
abstract storagetype Person {
   readonly state string name;
   state CORBA::OctetSeq photo;
};
```

is mapped to:

```
// Java
public interface Person
    extends CosPersistentState.StorageObject {
   public String name();
   public byte[] photo();
   public byte[] photo(CosPersistentState.ForUpdate fu);
   public void photo(byte[] new_one);
}
```

4.2.4 Storagetype Operations

Const and non-const operations on abstract and concrete storagehomes are mapped to public Java methods.

Table 4-1 shows the mapping for parameters of type S and ref<S> (where S is an abstract storagetype) For IDL parameters, the regular IDL to Java mapping is used.

Table 4-1 Mapping for PSDL parameters

PSDL parameter	Java parameter
in S param	`S param`
inout S param	`SHolder param`
out S param	`SHolder param`
(return) S	`(return) S`
in ref<S> param	`byte[] param`
inout ref<S> param	`byte[] param`
out ref<S> param	`byte[] param`
(return) ref<S>	`(return) byte[]`

4.2.5 Abstract Storagehomes

The mapping for PSDL abstract storagehomes is similar to the mapping for IDL local interfaces.

An abstract storagehome definition is mapped to a public Java interface with the same name. The mapped Java interface extends the mapped interfaces of all the abstract storagehomes inherited by this abstract storagehome. If an abstract storagehome does not extend any other abstract storagehome, its mapped interface extends the interface org.omg.CosPersistentState.StorageHomeBase.

4.2.6 Storagehome Operations

Operations on abstract and concrete storagehomes are mapped like non-const operations on storagetypes (see Section 4.2.4).

Note that key and factory operations are mapped as equivalent regular operations, as defined by Section 3.2.5.

4.2.7 Storagetype

A storagetype is mapped to a Java class with the same name. This class implements the mapped interfaces of all the abstract storagetypes implemented by the storagetype, and extends the mapped class of its base storagetype, if there is one. This class also provides a public default constructor.

If any of the abstract storagetypes implemented by the storagetype declares an operation, then the mapped class is abstract and public.

All state members implemented directly by the storagetype are mapped to public final accessor and modifier methods. The PSS implementation must be able to implement these methods without additional input from the developer.

For example:

```
abstract storagetype Dictionary {
    readonly state string from_language;
    readonly state string to_language;
    void insert(in string word, in string translation);
    string translate(in string word);
};

// a portable implementation:

struct Entry {
    string from;
    string to;
};
typedef sequence<Entry> EntryList;

storagetype PortableDictionary implements Dictionary {
    state EntryList entries;
};
```

is mapped to:

```
// Java
public abstract class PortableDictionary
   implements Dictionary /* ... */ {
   public final string from_language() { /* ... */ }
   public final string to_language() { /* ... */ }
   public final Entry[] entries() { /* ... */}
   public final Entry[] entries(ForUpdate fu) { /* ... */}
   public final void entries(Entries e) { /* ... */}
   public PortableDictionary() { /* ... */ }
      // ...
}
```

4.2.7.1 Storagehomes

A storage home is mapped to a Java class with the same name. This class implements the mapped interfaces of all the abstract storagehomes implemented by the storagehome, and extends the mapped class of its base storagehome, if there is one. This class also provides a public default constructor.

If any of the abstract storagehomes implemented by the storagehome declares an operation, then the mapped class is abstract and public.

A storagehome class implements all finder operations implicitly defined by abstract storagehomes directly implemented by the storagehome.

The mapped Java class provides four public non-abstract **`_create()`** methods:

- one that takes a parameter for each of its storagetype's state members and returns an incarnation
- one that takes a parameter for each of its storagetype's reference representation member, or no parameter of its storagetype has no reference representation.
- one that takes a parameter for each of the its storagetype's state members, plus a **`CosPersistentState.YieldRef`** parameter and returns a reference.
- one that takes a parameter for each of its storagetype's reference representation member (nothing if its storagetype has no reference representation), plus a **`CosPersistentState.YieldRef`** parameter and returns a reference.

The order of the _create() parameters is as follows: it begins with the base type of the storage type, proceed with the leftmost implemented abstract storage type and end with the state members defined in the storage type itself.

For example:

```
abstract storagetype Book {
   readonly state string title;
   state float price;
};
abstract storagehome BookStore of Book {};

storagetype PortableBook implements Book {};
storagehome PortableBookStore of PortableBook implements BookStore {};
```

maps to:

```
// Java
class PortableBookStore implements BookStore /* ... */ {
   public PortableBook _create(String name, float price) {/* ... */ }
   public PortableBook _create() {/* ... */}
   public byte[] _create(
                                String name,
                                float price,
                                CosPersistentState.YieldRef yr
                              ) {/* ... */}
   public byte[] _create(CosPersistentState.YieldRef yr)
                          {/* ... */}

   // ...
}
```

4.2.8 Factory Native Types

All the factory native types (**StorageObjectFactory**, **StorageHomeFactory**, **SessionFactory** and **SessionPoolFactory**) are mapped to the Java class **`java.lang.Class`**.

4.3 C++ Mapping

4.3.1 Abstract Storagetypes

An abstract storagetype definition is mapped to a C++ abstract base class with the same name; an abstract storagetype definition also results in the declaration of a C++ concrete class with “Ref” appended to its name, and the definition of _var and _out classes for memory management.

The mapped C++ class inherits (with public virtual inheritance) from the mapped classes of all the abstract storagetype inherited by this abstract storagetype. It also provides two public static member functions:

- **`_duplicate()`**: increases the reference count of the given parameter (if not null) and then returns itself
- **`_downcast()`**: like **`_downcast()`** for valuetypes.

For example:

```
// PSDL
abstract storagetype A {}; // implicitly inherits
                         // CosPersistentState::StorageObject
abstract storagetype B : A {};
```

is mapped to:

```
// C++
class  A  :
   public virtual CosPersistentState::StorageObject {};
class ARef :
   public virtual CosPersistentState::StorageObjectRef
{ /* ... */};
class A_var { /*... */};
class ARef_var {/* ... */};
class A_out { /*... */};
class ARef_out {/* ... */};

class B : public virtual A {};
class BRef {/*... */};
class B_var { /*... */};
class BRef_var {/* ... */};
class B_out { /*... */};
class BRef_out {/* ... */};
```

The forward declaration of a abstract storagetype is mapped to the forward declaration of its mapped class and “Ref” class.

The Ref class is a concrete C++ class which provides:

- a public default constructor that creates a null reference
- a non-explicit constructor which takes an incarnation of the target storage type.
- a public copy constructor
- a public destructor
- a public assignment operator
- a public assignment operator which takes an incarnation of the target abstract storage type.
- a public operator->() that dereferences this reference and returns the target object. The caller is not supposed to release this incarnation.
- a public deref() function which behaves like operator->()
- a public release() function which releases this reference
- a public destroy_object() function which destroys the target object
- a public get_pid() function which returns the pid of the target object.
- a public get_short_pid() function which returns the short-pid of the target object.
- a public is_null() function; it returns true if and only if this reference is null.
- a public get_storage_home() function which returns the storage home of the target object. This function increases the reference count of the return storage home.
- a conversion operator for each abstract storage type from which the corresponding abstract storage type derives directly or indirectly.
- a public typedef _target_type that type-defs the corresponding abstract storagetype.

- a public static _duplicate member function
- a public static _downcast member function

CosPersistentState::StorageObjectRef is a regular Ref class:

```
namespace CosPersistentState {
   class StorageObjectRef
   {
      public:
      typedef StorageObject _target_type;

      StorageObjectRef(
         StorageObject*  obj       = 0
      ) throw();

      StorageObjectRef(
         const StorageObjectRef& ref
      ) throw();

      StorageObjectRef&
      operator=(
         const StorageObjectRef& ref
      ) throw();

      StorageObjectRef&
      operator=(
         StorageObject* obj
      ) throw();

      void
      release() throw();

      StorageObject*
      deref() throw (CORBA::SystemException);

      StorageObject*
      operator->() throw (CORBA::SystemException);
      // not const!

      void
      destroy_object() throw (CORBA::SystemException);

      Pid*
      get_pid() const throw (CORBA::SystemException);

      ShortPid*
      get_short_pid() const throw (CORBA::SystemException);

      CORBA::Boolean
      is_null() const throw();
      StorageHomeBase_ptr
      get_storage_home() const
      throw (CORBA::SystemException);

      static StorageObjectRef
      _duplicate(
         StorageObjectRef ref
      );

      static StorageObjectRef
      _downcast(
         StorageObjectRef ref
      );

   // ...
   };
}
```

Note – C++ namespaces are used in this specification to represent mapped IDL modules. Depending on the target C++ compiler and ORB implementation, a module can be mapped to a C++ namespace, a class or a prefix.

The class **CosPersistentState::StorageObject** declares two pure virtual functions for reference counting, **_add_ref()** and **_remove_ref()**, and inherits from **CosPersistentState::StorageObjectBase**:

```
namespace CosPersistentState {

   class StorageObject : public virtual StorageObjectBase {
   public:

      virtual void _add_ref() = 0;
      virtual void _remove_ref() = 0;

      // normal mapping of PSDL operations:
      virtual void destroy_object()
            throw (SystemException) = 0;

      virtual Boolean object_exists()
            throw (SystemException) = 0;

      virtual Pid* get_pid() throw (SystemException) = 0;
      virtual ShortPid* get_short_pid()
             throw (SystemException) = 0;

      virtual StorageHomeBase* get_storage_home()
               throw (SystemException) = 0;
      static StorageObject*
      _duplicate(StorageObject*);

      static StorageObject*
      _downcast(StorageObject*);

   protected:
      virtual ~StorageObject()  {}
   };
}
```

The **CosPersistentState** namespace also provides two overloaded **release()** functions, one that takes a **StorageObject*** and releases a reference count if it is not null, and one that takes a **StorageObjectRef** and releases a reference count if it is not null.

4.3.2 Ref_var Classes

The _var class associated with a Ref class provides the same member functions as the Ref class (**operator->()**, **deref()**, **destroy_object(), get_pid(), get_short_pid(), is_null() and get_storage_home()**) with the same behavior, a constructor and an assignment operator that accepts a Ref object, a copy constructor and an assignment operator that accepts a const Ref-var object reference, an const **in()** member function that returns a Ref object, an non-const **inout()** member function that returns a non-const Ref object reference, an **out()** member function that returns a non-const Ref object reference, and a **_retn()** member function that returns a Ref object, releasing the Ref held by this var object.

4.3.3 Arrays and Sequences

The C++ mapping for sequences and arrays of abstract storagetypes/references to abstract storagetype is like the C++ mapping for sequences and arrays of IDL types.

4.3.4 State Members

Each state member is mapped to a number of overloaded public pure virtual accessor and modifier functions, with the same name as the state member. These functions can raise any CORBA standard exception.

A state member whose C++ type is a basic type is mapped like a value data member. There is no modifier function if the state member is read-only.

For example:

```
// PSDL
abstract storagetype Person {
   state string name;
};
```

is mapped to

```
// C++
class Person : public virtual StorageObject {
public:
   virtual const char* name() const = 0;
   virtual void name(const char* s) = 0;  // copies
   virtual void name(char* s) = 0;        // adopts
   virtual void name(String_var& s) = 0; // adopts
};
```

A state member whose type is an abstract storagetype is mapped to a read-only accessor, a read-write accessor and a modifier, or just a read-only accessor when the state member is read-only.

For example:

```
// PSDL
abstract storagetype A {};
abstract storagetype B {
   state A embedded;
};
```

is mapped to:

```
// C++
class B : public virtual StorageObject {
public:
   virtual const A& embedded() const = 0;
   virtual A& embedded(CosPersistentState::ForUpdate) = 0;
   virtual void embedded(const A&) = 0; // copies
};
```

A state member whose type is a reference to an abstract storagetype is mapped to two accessors and one modifier functions. One of the accessor functions takes no parameter and returns a storage object incarnation, the other takes a CosPersistentState::YieldRef parameter and returns a reference. The modifier function accepts a reference object. If the state member is read-only, only the accessor functions are generated. For example:

```
abstract storagetype Bank;

abstract storagetype Account {
   state long id;
   state ref<Bank> my_bank;
};
```

is mapped to:

```
// C++
class Account : public virtual StorageObject {
public:
   virtual CORBA::Long id() = 0;
   virtual void id(CORBA::Long l) = 0;
   virtual Bank* my_bank() const= 0;
   virtual BankRef my_bank(CosPersistentState::YieldRef yr)
const = 0;
   virtual void my_bank(BankRef b) = 0;
};
```

All other state members are mapped to two accessor functions (one read-only, one read-write) and one modifier function. If such a state member is read-only, only the read-only accessor is generated. For example:

```
abstract storagetype Person {
   readonly state string name;
   state CORBA::OctetSeq photo;
};
```

is mapped to:

```
// C++
class Person : public virtual StorageObject {
public:
   virtual const char* name() = 0;
   virtual OctetSeq* photo() const = 0;
   virtual OctetSeq* photo(CosPersistentState::ForUpdate fu)
                = 0;
   virtual void photo(const OctetSeq& new_one) = 0;
};
```

4.3.5 Storagetype Operations

A const operation on an abstract or concrete storagetype is mapped to a const virtual public member function; a non-const operation on an abstract or concrete storagetype is mapped to a non const virtual public member function.

Table 4-2 shows the mapping for parameters of type S and ref<S> (where S is an abstract storagetype). For IDL parameters, the regular IDL to C++ mapping is used .

Table 4-2 Mapping for PSDL parameters

PSDL parameter	C++ parameter
in S param	`const S& param`
inout S param	`S& param`
out S param	`S_out param`
(return) S	`(return) S*`
in ref<S> param	`SRef param`
inout ref<S> param	`SRef& param`
out ref<S> param	`SRef_out param`
(return) ref<S>	`(return) SRef`

References are always passed by value or reference (never through pointers).

4.3.6 Abstract Storagehomes

The mapping for PSDL abstract storagehomes is similar to the mapping for IDL local interfaces.

An abstract storagehome definition is mapped to a C++ class with the same name. The mapped C++ class inherits using public virtual interitance from the mapped classes of all the abstract storagehomes inherited by this abstract storagehome. If an abstract storagehome does not extend any other abstract storagehome, its mapped class inherits (using public virtual inheritance) from CosPersistentState::StorageHomeBase.

Like with local interfaces, the mapped class has associated _var and _out helper classes.

4.3.7 Storagehome Operations

Operations on abstract and concrete storagehomes are mapped like non-const operations on storagetypes (see Section 4.3.5).

Note that key and factory operations are mapped as equivalent regular operations, as defined by Section 3.2.5.

4.3.8 Storagetype

A storagetype is mapped to a C++ class with the same name. This class inherits from the mapped classes of all the abstract storagetypes implemented by the storagetype, and from the mapped class of its base storagetype, if there is one. This class also provides a public default constructor.

All state members implemented directly by the storagetype are implemented by the mapped class, as public functions. The PSS implementation must be able to implement these functions without additional input from the developer.

For example:

```
abstract storagetype Dictionary {
    readonly state string from_language;
    readonly state string to_language;
    void insert(in string word, in string translation);
    string translate(in string word);
};

// a portable implementation:

struct Entry {
    string from;
    string to;
};
```

```
typedef sequence<Entry> EntryList;

storagetype PortableDictionary implements Dictionary {
      state EntryList entries;
};
```

is mapped to:

```
// C++
class PortableDictionary : public virtual Dictionary /* ... */ {
public:
      const char* from_language() const;
      const char* to_language() const;
      EntryList* entries() const;
      EntryList* entries(CosPersistentState::ForUpdate fu);
      void entries(const EntryList&);
      PortableDictionary();
      // ...
};
```

For each storagetype, a concrete “Ref” class is also generated. Like the Ref class generated for an abstract storage type, it provides

- a public default constructor that creates a null reference
- a non-explicit constructor which takes an incarnation of the target storage type.
- a public copy constructor
- a public destructor
- a public assignment operator
- a public assignment operator which takes an incarnation of the target storage type.
- a public operator->() that dereferences this reference and returns the target object. The caller is not supposed to release this incarnation.
- a public deref() function which behaves like operator->()
- a public release() function which releases this reference
- a public destroy_object() function which destroys the target object
- a public get_pid() function which returns the pid of the target object.
- a public get_short_pid() function which returns the short-pid of the target object.
- a public is_null() function; it returns true if and only if this reference is null.
- a public get_storage_home() function which returns the storage home of the target object. This function increases the reference count of the return storage home.
- a conversion operator for each abstract storage type implemented by the corresponding storage type (directly or indirectly).
- a conversion operator for each storage type implemented from which the corresponding storage type derives (directly or indirectly).

- a public typedef _target_type that type-defs the corresponding storagetype.
- a public static _duplicate member function
- a public static _downcast member function

4.3.9 Storagehomes

A storagehome is mapped to a C++ class with the same name. This class inherits from the mapped classes of all the abstract storagehomes implemented by the storagehome, and from the mapped class of its base storagehome, if there is one. This class also provides a public default constructor.

A storagehome class implements all finder operations implicitly defined by abstract storagehomes directly implemented by the storagehome.

The mapped C++ class provides two public non-virtual **`_create()`** member functions:

- one that takes a parameter for each of the its storagetype's state members and returns an incarnation
- one that takes a parameter for each of the its storagetype's state members, plus a **CosPersistentState::YieldRef** parameter and returns a reference.

It also provides two public virtual **`_create()`** member functions:

- one that takes a parameter for each of the its storagetype's reference representation members (no parameter if the storagetype has no reference representation) and returns an incarnation
- one that takes a parameter for each of the its storagetype's reference representation members (nothing if the storagetype has no reference representation), plus a **CosPersistentState::YieldRef** parameter and returns a reference.

The order of the _create() parameters is as follows: it begins with the base type of the storage type, proceed with the leftmost implemented abstract storage type and end with the state members defined in the storage type itself.

Like other mapped types, this class also provides a public static **`_duplicate()`** and a public static **`_downcast(`**) member function.

For example:

```
abstract storagetype Book  {
    readonly state string title;
    state float price;
};
abstract storagehome BookStore of Book {};

storagetype PortableBook implements Book {};
storagehome PortableBookStore of PortableBook implements BookStore {};
```

maps to:

```
// C++
class PortableBookStore : public virtual BookStore /* ... */
{
public:
   PortableBook* _create(const char* name, Float price);
   PortableBook* _create();
   PortableBookRef _create(
                        const char* name,
                        Float price,
                        CosPersistentState::YieldRef yr
                     );
   PortableBookRef _create(
                           CosPersistentState::YieldRef yr
                     );
   // ...
};
```

4.3.10 Factory Native Types

The native factory types **StorageObjectFactory**, **StorageHomeFactory**, **SessionFactory** and **SessionPoolFactory** map the C++ classes with the same names, defined as follows:

```
namespace CosPersistentState {

   template class<T>
   class Factory {
      public:
         virtual T* create() throw (SystemException) = 0;
         virtual void _add_ref() {}
         virtual void _remove_ref() {}
         virtual ~Factory() {}
   };

   typedef Factory<StorageObject> StorageObjectFactory;
   typedef Factory<StorageHomeBase> StorageHomeFactory;
   typedef Factory<Session> SessionFactory;
   typedef Factory<SessionPool> SessionPoolFactory;

}
```

CosPersistentState Module *A*

A.1 Complete IDL

```
//File: CosPersistentState.psdl

// Copyright 1998-1999 by the Object Management Group.
// All Rights Reserved.

#ifndef _COS_PERSISTENT_STATE_PSDL_
#define _COS_PERSISTENT_STATE_PSDL_

#include <orb.idl>
#include <CosTransactions.idl>

module CosPersistentState {

   local interface CatalogBase;
   local interface Connector;
   local interface EndOfAssociationCallback;
   local interface Session;
   local interface SessionPool;
   local interface StorageHomeBase;
   local interface TransactionalSession;

   native StorageObjectBase;
   native StorageObjectFactory;
   native StorageHomeFactory;
   native SessionFactory;
   native SessionPoolFactory;

   exception NotFound {};

   typedef string TypeId;
   typedef CORBA::OctetSeq Pid;
   typedef CORBA::OctetSeq ShortPid;


   abstract storagetype StorageObject {
       void destroy_object();
       boolean object_exists();
       Pid get_pid();
       ShortPid get_short_pid();
       StorageHomeBase get_storage_home();
   };


   enum YieldRef { YIELD_REF };
   enum ForUpdate { FOR_UPDATE };

   typedef short IsolationLevel;
   const IsolationLevel READ_UNCOMMITTED  = 0;
   const IsolationLevel READ_COMMITTED    = 1;
   const IsolationLevel REPEATABLE_READ   = 2;
   const IsolationLevel SERIALIZABLE      = 3;

   typedef short TransactionPolicy;
   const TransactionPolicy NON_TRANSACTIONAL = 0;
   const TransactionPolicy TRANSACTIONAL     = 1;

   typedef short AccessMode;
   const AccessMode READ_ONLY = 0;
   const AccessMode READ_WRITE = 1;

   struct Parameter {
       string name;
       any val;
   };

   typedef sequence<Parameter> ParameterList;

   typedef sequence<TransactionalSession> TransactionalSessionList;


   //------------------------------------------------------------
   // Connector
   //------------------------------------------------------------

   local interface Connector {

       readonly attribute string implementation_id;

       Pid get_pid(in StorageObjectBase obj);
       ShortPid get_short_pid(in StorageObjectBase obj);
```

```
Session
create_basic_session(
   in AccessMode access_mode,
   in ParameterList additional_parameters
);

TransactionalSession
create_transactional_session(
   in AccessMode access_mode,
   in IsolationLevel default_isolation_level,
   in EndOfAssociationCallback callback,
   in ParameterList additional_parameters
);

SessionPool
create_session_pool(
   in AccessMode access_mode,
   in TransactionPolicy tx_policy,
   in ParameterList additional_parameters
);

TransactionalSession current_session();

TransactionalSessionList
sessions(
   in CosTransactions::Coordinator transaction
);

StorageObjectFactory
register_storage_object_factory(
    in TypeId storage_type_name,
    in StorageObjectFactory storage_object_factory
);

StorageHomeFactory
register_storage_home_factory(
    in TypeId storage_home_type_name,
    in StorageHomeFactory storage_home_factory
);

SessionFactory
register_session_factory(
  in SessionFactory session_factory
);

SessionPoolFactory
register_session_pool_factory(
    in SessionPoolFactory session_pool_factory
);
```

```
};


//-----------------------------------------------------------
// CatalogBase
//-----------------------------------------------------------

local interface CatalogBase {

   readonly attribute AccessMode access_mode;

   StorageHomeBase
   find_storage_home(in string storage_home_id)
      raises (NotFound);

   StorageObjectBase
   find_by_pid(in Pid the_pid) raises (NotFound);

   void flush();
   void refresh();

   void free_all();

   void close();
};


//-----------------------------------------------------------
// StorageHomeBase
//-----------------------------------------------------------

local interface StorageHomeBase {

   StorageObjectBase
   find_by_short_pid(in ShortPid short_pid)
         raises (NotFound);

   CatalogBase get_catalog();
};


//-----------------------------------------------------------
// Session
//-----------------------------------------------------------

local interface Session : CatalogBase {};

//-----------------------------------------------------------
// TransactionalSession
//-----------------------------------------------------------
```

```
 local interface TransactionalSession : Session {

     readonly attribute IsolationLevel default_isolation_level;

     typedef short AssociationStatus;
     const AssociationStatus NO_ASSOCIATION = 0;
     const AssociationStatus ACTIVE        = 1;
     const AssociationStatus SUSPENDED     = 2;
     const AssociationStatus ENDING        = 3;

     void start(in CosTransactions::Coordinator transaction);
     void suspend(in CosTransactions::Coordinator transaction);
     void end(
          in CosTransactions::Coordinator transaction,
          in boolean success
        );

     AssociationStatus get_association_status();

     CosTransactions::Coordinator transaction();
 };

 local interface EndOfAssociationCallback {
     void released(in TransactionalSession session);
 };

 //-------------------------------------------------------------
 // SessionPool
 //-------------------------------------------------------------

 typedef sequence<Pid> PidList;

 local interface SessionPool : CatalogBase {

     void flush_by_pids(in PidList pids);
     void refresh_by_pids(in PidList pids);

     readonly attribute TransactionPolicy transaction_policy;
 };

};

#endif // _COS_PERSISTENT_STATE_PSDL_
```

Example: An Implementation of the Naming Service B

B.1 Introduction

This non-normative section presents a simple, portable implementation of the Naming Service using the Persistent State Service.

The Naming Service defines only two IDL interfaces, **NamingContext** and **BindingIterator**.

From the Naming Service chapter: “A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique.”

A naming context is typically a *persistent* CORBA object: its implementation outlives the process that created it, by storing information (its state, or at least part of its state) in a datastore.

A binding iterator is an object used to iterate over the content of a naming context; binding iterators are *transient* objects created by naming contexts.

B.1.1 Specifying Storage Objects

We choose to associate with each naming context a storage object that represents its state. This state is a set of name bindings -- which can also be viewed as a name-to-object-reference map; so we specify our naming context state storage objects as follows:

```
// file NamingContextState.psdl
#include <CosNaming.idl>
abstract storagetype NamingContextState {
    Object resolve(in CosNaming::NameComponent n)
        raises(CosNaming::NotFound, CosNaming::CannotProceed,
                CosNaming::InvalidName);
    void bind(in CosNaming::NameComponent n, in Object obj)
        raises(CosNaming::NotFound, CosNaming::CannotProceed,
                CosNaming::InvalidName, CosNaming::AlreadyBound);
    void rebind(in CosNaming::NameComponent n, in Object obj)
        raises(CosNaming::NotFound, CosNaming::CannotProceed,
                CosNaming::InvalidName);
    void unbind(in CosNaming::NameComponent n, in Object obj)
        raises(CosNaming::NotFound, CosNaming::CannotProceed,
                CosNaming::InvalidName);

    boolean is_empty();

    CosNaming::BindingIterator create_iterator();
};

abstract storagehome NamingContextStateHome of NamingContextState {
    factory create();
};
```

B.1.2 Implementing NamingContext Servants

The implementation of a naming context servant is quite simple: most operations simply “unwrap” a name by calling resolve to reduce its length to one. When the name’s length is one, the implementation delegates the real work to its storage object.

To establish the association naming context object -- storage object, we choose to use the storage object short pid as the naming context object’s object id. Further, for simplicity (and scalability), we use a single servant to handle all requests to NamingContext objects.

In this first cut, we use the same storage home instance (and hence catalog) to access all storage objects.

```
// POA-based Java servant class

public class NamingContextImpl extends POA_NamingContext {

   // fields
   private NamingContextStateHome m_home;
   private POA m_poa;

   // helper methods

   NamingContextState my_state() {
      return (NamingContextState)
         my_home().find_by_short_pid(_object_id());
   }
```

```
NamingContextStateHome my_home() {
   // first cut: all naming contexts use the same
   // storage home instance
   return m_home;
}

// Constructor

NamingContextImpl(
      NamingContextStateHome home,
      POA poa
)
{
   m_home = home;
   m_poa = poa;
}

// implementation of IDL operations

public CORBA.Object
resolve(NameComponent[] n) {
   if (n.length == 1) {
      return my_state().resolve(n[0]);
   }
   else {
      NamingContext new_target = NamingContext.narrow(
               my_state().resolve(n[n.length -1])
            );
      NameComponent[] rest_of_name
         = new NameComponent[n.length - 1];
      System.arraycopy(n,0,rest_of_name,0,n.length-1);
      return new_target.resolve(rest_of_name);
   }
}

// similar implementations for bind, unbind, rebind,
// bind_context, rebind_context

public NamingContext new_context() {
   NamingContextState ref = my_home().create();
   return NamingContext.narrow(
         _poa().create_reference_with_id(
            ref.short_pid(),
            “IDL:omg.org/CosNaming/NamingContext:1.0”
         )
      );
}

// bind_new_context() simply creates a new context and
// binds it
```

```
   public void destroy() throws NoEmpty {
      if (my_state().is_empty()) {
         my_state().destroy();
      }
      else {
         throw new NotEmpty;
      }
   }

   // list() 'wraps' my_state().create_iterator()

}
```

B.1.3 Implementing Storage Objects

Depending on the PSS implementation, and the underlying datastore, NamingContextState can be implemented in different ways.

A PSS implementation for an ODMG system would probably provide a **dictionary** type, as a proprietary extension. For example:

```
storagetype OdmgNCtxState implements NamingContextState {
   state dictionary<CosNaming::ComponentName, Object> m_map;
};
storagehome OdmgNCtxStateHome of OdmgNCtxState implements
   NamingContextStateHome {};
```

A PSS implementation for Oracle8 could provide a nested table type, since it is a native Oracle8 feature. For example:

```
nestedtable BindingTable {
   state CosNaming::ComponentName name;
   state Object obj;
   key name;
};
storagetype OracleNCtxState implements NamingContextState {
   state BindingTable m_table;
};
storagehome OracleNCtxStateHome of OracleNCtxState implements
   NamingContextStateHome {};
```

However, for some developers, portability is more important than performance. Such a developer would use only standard PSDL to define his/her implementation, PortableNamingContextState:

```
struct ListElement {
   CosNaming::NameComponent name;
   Object obj
};
typedef sequence<ListElement> List;
```

```
storagetype PortableNCtxState implements NamingContextState {
   state List m_list;
};

storagehome PortableNCtxStateHome of PortableNCtxState
      implements NamingContextStateHome
{};

// Java implementation

public class PortableNCtxStateImpl extends PortableNCtxState
{
   public CORBA.Object resolve(NameComponent n)
      throws NotFound, CannotProceed, InvalidName
   {
      for (int i = 0; i < m_list.length; i++)
      {
         if ((m_list[i].name.id == n.id)
            && (m_list[i].name.kind = n.kind))
         {
             return m_list[i].obj;
         }
      }
   }

   public Boolean is_empty() {
      return (m_list.length == 0);
   }

   // etc.
}

public class PortableNCtxStateHomeImpl
         extends PortableNCtxStateHome {
   public NamingContextState create() {/* generated */)
}
```

B.1.4 Completing the Naming Server

Now that our servants and storage objects are implemented, we need to create the ‘main’ of our naming server:

- Get the PSS connector, to register our incarnation and storage home incarnation factories.
- Get the root POA, to create a child POA with the following policies: PERSISTENT, USE_DEFAULT_SERVANT, USER_ID, MULTIPLE_ID, NON_RETAIN.
- Create a session and find the storage home that manages our storage objects.
- Create a servant with this ‘persistent’ POA and this storage home incarnation.

- If the server is run for the first time, create the root naming context and prints its object reference.

```
public class NamingServer {
   public static void main(String[] args) {

      // initializes ORB
      CORBA.ORB myOrb = CORBA.ORB.init(args);

      // get connector registry
      CosPersistentState.ConnectorRegistry registry
         = CosPersistentState.ConnectorRegistryHelper.
            narrow(
               myOrb.resolve_initial_references(“PSS”)
            );
      // get connector
      CosPersistentState.Connector connector =
         registry.find_connector(“”);

      // register storage object factory
      connector.register_storage_object_factory(
         “PortableNCtxState”,
         Class.forName(“PSDL:PortableNCtxStateImpl:1.0”)
      );

      // register storage home factory
      connector.register_storage_home_factory(
         “PSDL:PortableNCtxStateHome:1.0”,
         Class.forName(“PortableNCtxStateHomeImpl”)
      );

      // create session
      CosPersistentState.Session mySession
         = connector.create_basic_session(
               org.omg.CosPersistentState.READ_WRITE,
               “”,
               parameters
         );

      // get storage home
      NamingContextStateHome home = (NamingContextStateHome)
         mySession.find_storage_home(
               “PSDL:PortableNCtxStateHomeImpl:1.0”
               );

      // get root POA
      PortableServer.POA rootPOA
         = PortableServer.POAHelper.narrow(
            myOrb.resolve_initial_references(“RootPOA”)
      );
      // create policies
      CORBA.Policy policies[5];
      policies[0] = rootPOA.create_lifespan_policy(
         PortableServer.LifespanPolicyValue.PERSISTENT
      );
      policies[1] =rootPOA.create_request_processing_policy(
         PortableServer.RequestProcessingPolicyValue.
            USE_DEFAULT_SERVANT
      );
      policies[2] = rootPOA.create_id_uniqueness_policy(
         PortableServer.IdUniquenessPolicyValue.MULTIPLE_ID
      );
      policies[3] = rootPOA.create_id_assignment_policy(
         PortableServer.IdAssignmentPolicyValue.USER_ID
      );
      policies[4] = rootPOA.create_servant_retention_policy(
         PortableServer.ServantRetentionPolicyValue.
            NON_RETAIN
      );

      // create POA for naming contexts
      PortableServer.POAManager poaMgr
         = rootPOA.the_POAManager();
      PortableServer.POA poa = rootPOA.create_POA(
         “Naming”, null, policies
      );

      // the first time, create a root naming context
      // and prints its IOR
      if (firstTime) {
         byte[] root_id = home.create().short_pid();
         CORBA.Object root_naming_context =
             poa.create_reference_with_id(
               root_id,
               “IDL:omg.org/CosNaming/NamingContext:1.0”
            );
         System.out.println(
            myOrb.object_to_string(root_naming_context)
         );
      }

      // create and set servant
      NamingContextImpl servant(home, poa);
      poa.set_servant(servant);

      // start server
      poaMgr.activate();
      myOrb.run();
      mySession.close();
   }
```

B.1.5 A Transactional Naming Server

Our first naming server is non-transactional: we created a basic session, and used the same storage home incarnation for all requests.

We can easily upgrade it to a transactional naming server, by updating the **`my_home()`** method and the constructor of the **`NamingContextImpl`** servant class:

```
NamingContextStateHome my_home() {
   return (NamingContextStateHome) m_connector.
      current_session().find_storage_home(m_home_name);
}

NamingContextImpl(
   String home_name,
   POA poa
)
{
   m_home_name = home_name;
   m_poa = poa;
}
```

This also assumes that we have a mechanism that deals with the association between OTS transactions and sessions. For example:

```
// The implementation of MySessionPool creates/manages
// transactional sessions; it registers a
// EndOfAssociationCallback to be notified when a session
// is released by the PSS implementation.

public interface MySessionPool {
                     TransactionalSession
get_idle_session();
}

public class AssociationManager {

   private CosTransactions.Current m_txcurrent;
   private MySessionPool m_pool;

   // Somehow called before the business logic of each
   // operation
   public void start_of_request() {
      CosTransactions.Control control =
          m_txcurrent.get_control();
      if (control != null) {
         CosPersistentState.TransactionalSession
            session = m_connector.current_session();
         if (session == null){
            session = m_pool.get_idle_session();
         }
      }
      session.start(control.get_coordinator());
   }

   // Somehow called after the business logic of each
   // operation
   public void end_of_request() {
      CosTransactions.Control control =
         m_txcurrent.get_control();
      if (control != null) {
         m_connector.current_session().suspend(
            control.get_coordinator()
         );
      }
   }
}
```

Relationship to Other Services C

C.1 Introduction

This appendix describes the relationship between the Persistent State Service and the other Common Object Services defined by the OMG.

C.1.1 Transaction Service

The Persistent State Service relies on the Transaction Service for transactions. The relationship with this service is fully described in “Accessing Storage Objects”.

C.1.2 Security Service

This section specifies how a Persistent State Service implementation fits into the overall CORBA Security framework. The Security Service provides means to secure interactions between CORBA clients and CORBA Objects; the Persistent State Service provides a service to servant developers and is not directly involved in any CORBA Object invocation. As a result, there is no overlap of functionality in these two services.

Nonetheless, a Persistent State Service implementation can help application developers take advantage of the security features provided by their datastore to implement secure CORBA applications. Such a security-aware implementation shall support the general model described below.

Model

Storage object provided by a Persistent State Service implementation, and CORBA Objects managed by an ORB and a Security Service implementation are in different security policy domains, and generally in different security technology domains.

Conceptually, the operations **Connector::create_basic_session, Connector::create_transactional_session** and **Connector::create_session_pool** perform a **SecurityLevel2::PrincipalAuthenticator::authenticate** call. Some compliant implementations may have drastic restrictions: for example, a simple file-system based implementation can support only one principal per process with authentication performed by the operating system.

Conformance Requirements *D*

A compliant implementation must implement the **CosPersistentState** module entirely in at least one programming language for which this specification defines a mapping. It must also provide a tool that reads PSDL specifications and generate code in this programming language.

There are two optional features: transaction support and transparent persistence.

The operation **create_transactional_session** on the connector of an implementation that does not support transactions must raise the NO_IMPLEMENT standard exception. The operation **create_session_pool** of an implementation that does not support transactions must raise the NO_IMPLEMENT standard exception when the transaction_policy parameter is **TRANSACTIONAL**. A compliant implementation that supports transactions as specified in this specification can claim to be “a compliant Persistent State Service implementation with transaction support”.

A compliant implementation that supports transparent persistence can claim to be “a compliant Persistent State Service implementation with transparent persistence support.”

References E

[ODMG] Rick G. G. Cattell et al, *The Object Database Standard: ODMG 2.0*, The Morgan Kaufmann Series in Data Management Systems, 1997

[SQL3] *ISO Working Draft, Database Language SQL -- Part 2: Foundation (SQL/Foundation),* September 1998

[XA] *Distributed Transaction Processing: The XA Specification, X/Open Document C193*, X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.