Chapter 7. The Interface Repository Framework

Contents

Chapter 7. The Interface Repository Framework

7.1 Introduction

The SOM Interface Repository (IR) is a database that the SOM Compiler optionally creates and maintains from the information supplied in IDL source files. The Interface Repository contains persistent objects that correspond to the major elements in IDL descriptions. The SOM Interface Repository Framework is a set of classes that provide methods whereby executing programs can access these objects to discover everything known about the programming interfaces of SOM classes.

The programming interfaces used to interact with Interface Repository objects, as well as the format and contents of the information they return, are architected and defined as part of the Object Management Group's CORBA standard. The classes composing the SOM Interface Repository Framework implement the programming interface to the CORBA Interface Repository. Accordingly, the SOM Interface Repository Framework supports all of the interfaces described in *The Common Object Request Broker: Architecture and Specification* (OMG Document Number 91.12.1, Revision 1.1, chapter 7).

As an extension to the CORBA standard, the SOM Interface Repository Framework permits storage in the Interface Repository of arbitrary information in the form of SOM IDL **modifiers**. That is, within the SOM-unique **implementation** section of an IDL source file or through the use of the **#pragma modifier** statement, user-defined modifiers can be associated with any element of an IDL specification. (See the section entitled "SOM Interface Definition Language" in Chapter 4, "SOM IDL and the SOM Compiler.") When the SOM Compiler creates the Interface Repository from an IDL specification, these potentially arbitrary modifiers are stored in the IR and can then be accessed via the methods provided by the Interface Repository Framework.

This chapter describes, first, how to build and manage interface repositories, and second, the programming interfaces embodied in the SOM Interface Repository Framework.

7.2 Using the SOM Compiler to Build an Interface Repository

The SOMobjects Toolkit includes an Interface Repository emitter that is invoked whenever the SOM Compiler is run using an **sc** or **somc** command with the **–u** option (which “updates” the interface repository). The IR emitter can be used to create or update an Interface Repository file. The IR emitter expects that an environment variable, SOMIR, was first set to designate a file name for the Interface Repository. For example, to compile an IDL source file named “newcls.idl” and create an Interface Repository named “newcls.ir”, use a command sequence similar to the following:

For OS/2:

```
set SOMIR=c:\myfiles\newcls.ir
sc –u newcls
```

For AIX:

```
export SOMIR=~/newcls.ir
sc –u newcls
```

For Windows:
Note: Ensure that no spaces separate the environment variable “SOMIR”, the equals sign “=”, and the value being set.

```
set SOMIR=c:\myfiles\newcls.ir
somc –u newcls
```

If the SOMIR environment variable is not set, the Interface Repository emitter creates a file named “som.ir” in the current directory.

The **sc** or **somc** command runs the Interface Repository emitter plus any other emitters indicated by the environment variable SMEMIT (described in the topic “Running the SOM Compiler” in Chapter 4, “SOM IDL and the SOM Compiler”). To run the Interface Repository emitter by itself, issue the **sc** or **somc** command with the **–s** option (which overrides SMEMIT) set to “ir”. For example:

```
sc –u –sir newcls        (On OS/2 or AIX)
somc –u –sir newcls      (On Windows)
```

or equivalently,

```
sc –usir newcls          (On OS/2 or AIX)
somc –usir newcls        (On Windows)
```

The Interface Repository emitter uses the SOMIR environment variable to locate the designated IR file. If the file does not exist, the IR emitter creates it. If the named interface repository already exists, the IR emitter checks all of the “type” information in the IDL source file being compiled for internal consistency, and then changes the contents of the interface repository file to agree with with the new IDL definition. For this reason, the use of the –u compiler flag requires that all of the types mentioned in the IDL source file must be fully defined within the scope of the compilation. Warning messages from the SOM Compiler about undefined types result in actual error messages when using the –u flag.

The additional type checking and file updating activity implied by the –u flag increases the time it takes to run the SOM Compiler. Thus, when developing an IDL class description from scratch, where iterative changes are to be expected, it may be preferable *not* to use the –u compiler option until the class definition has stabilized.

7.3 Managing Interface Repository files

Just as the number of interface definitions contained in a single IDL source file is optional, similarly, the number of IDL files compiled into one interface repository file is also at the programmer's discretion. Commonly, however, all interfaces needed for a single project or class framework are kept in one interface repository.

The SOM IR file "som.ir"

The SOMobjects Toolkit includes an Interface Repository file ("som.ir") that contains objects describing all of the types, classes, and methods provided by the various frameworks of the SOMobjects Toolkit. Since all new classes will ultimately be derived from these predefined SOM classes, some of this information also needs to be included in a programmer's own interface repository files.

For example, suppose a new class, called "MyClass", is derived from **SOMObject**. When the SOM Compiler builds an Interface Repository for "MyClass", that IR will also include all of the information associated with the **SOMObject** class. This happens because the **SOMObject** class definition is inherited by each new class; thus, all of the **SOMObject** methods and typedefs are implicitly contained in the new class as well.

Eventually, the process of deriving new classes from existing ones would lead to a great deal of duplication of information in separate interface repository files. This would be inefficient, wasteful of space, and extremely difficult to manage. For example, to make an evolutionary change to some class interface, a programmer would need to know about and subsequently update all of the interface repository files where information about that interface occurred.

One way to avoid this dilemma would be to keep all interface definitions in a single interface repository (such as "som.ir"). This is not recommended, however. A single interface repository would soon grow to be unwieldy in size and become a source of frequent access contention. Everyone involved in developing class definitions would need update access to this one file, and simultaneous uses might result in longer compile times.

Managing IRs via the SOMIR environment variable

The SOMobjects Toolkit offers a more flexible approach to managing interface repositories. The SOMIR environment variable can reference an ordered list of separate IR files, which process from left to right. Taken as a whole, however, this gives the appearance of a single, logical interface repository. A programmer accessing the contents of "the interface repository" through the SOM Interface Repository framework would not be aware of the division of information across separate files. It would seem as though all of the objects resided in a single interface repository file.

A typical way to utilize this capability is as follows:

- The first (leftmost) Interface Repository in the SOMIR list would be "som.ir". This file contains the basic interfaces and types needed in all SOM classes.
- The second file in the list might contain interface definitions that are used globally across a particular enterprise.
- A third interface repository file would contain definitions that are unique to a particular department, and so on.
- The final interface repository in the list should be set aside to hold the interfaces needed for the project currently under development.

Developers working on different projects would each set their SOMIR environment variables to hold slightly different lists. For the most part, the leftmost portions of these lists would be the same, but the rightmost interface repositories would differ. When any given developer is ready to share his/her interface definitions with other people outside of the immediate work group, that person's interface repository can be promoted to inclusion in the master list.

With this arrangement of IR files, the more stable repositories are found at the left end of the list. For example, a developer should never need to make any significant changes to "som.ir", because these interfaces are defined by IBM and would only change with a new release of the SOMobjects Toolkit.

The Interface Repository Framework only permits updates in the rightmost file of the SOMIR interface repository list. That is, when the SOM Compiler –u flag is used to update the Interface Repository, only the final file on the IR list will be affected. The information in all preceding interface repository files is treated as "read only". Therefore, to change the definition of an interface in one of the more global interface repository files, a developer must overtly construct a special SOMIR list that omits all subsequent (that is, further to the right) interface repository files, or else petition the owner of that interface to make the change.

It is important that the rightmost filename in the SOMIR interface repository list *not* appear elsewhere in the list. For example, the following setting for SOMIR:

```
%SOMBASE%\ETC\SOM.IR;SOM.IR;C:\IR\COMPANY.IR;SOM.IR
```

would cause problems when attempting to update the SOM.IR file, because SOM.IR appears twice in the list.

Here is an example that illustrates the use of multiple IR files with the SOMIR environment variable. In this example, the SOMBASE environment variable represents the directory in which the SOMobjects Toolkit files have been installed. Only the "myown.ir" interface repository file will be updated with the interfaces found in files "myclass1.idl", "myclass2.idl", and "myclass3.idl".

For OS/2:

```
set BASE_IRLIST=%SOMBASE%\IR\SOM.IR;C:\IR\COMPANY.IR;C:\IR\DEPT10.IR
set SOMIR=%BASE_IRLIST%;D:\MYOWN.IR
set SMINCLUDE=.;%SOMBASE%\INCLUDE;C:\COMPANY\INCLUDE;C:\DEPT10\INCLUDE
sc –usir myclass1
sc –usir myclass2
sc –usir myclass3
```

For AIX:

```
export BASE_IRLIST=$SOMBASE/ir/som.ir:/usr/local/ir/company.ir:\
    /usr/local/ir/dept10.ir
export SOMIR=$BASE_IRLIST:~/myown.ir
export SMINCLUDE=.:$SOMBASE/INCLUDE:/usr/local/company/include:\
    /usr/local/dept10/include
sc –usir myclass1
sc –usir myclass2
sc –usir myclass3
```

For Windows:
The following example (of multiple IR files with the SOMIR environment variable) will work correctly only if it is executed from within a .BAT file. (Otherwise, the %BASE_IRLIST% is not interpreted but is taken literally.)

```
set BASE_IRLIST=%SOMBASE%\IR\SOM.IR;C:\IR\COMPANY.IR;C:\IR\DEPT10.IR
set SOMIR=%BASE_IRLIST%;D:\MYOWN.IR
set SMINCLUDE=.;%SOMBASE%\INCLUDE;C:\COMPANY\INCLUDE;C:\DEPT10\INCLUDE
somc –usir myclass1
somc –usir myclass2
somc –usir myclass3
```

Placing 'private' information in the Interface Repository

When the SOM Compiler updates the Interface Repository in response to the **–u** flag, it uses all of the information available from the IDL source file. However, if the __PRIVATE__ preprocessor variable is used to designate certain portions of the IDL file as private, the preprocessor actually removes that information before the SOM Compiler sees it. Consequently, private information will not appear in the Interface Repository unless the **–p** compiler option is also used in conjunction with **–u.** For example:

```
sc –up myclass1          (On AIX or OS/2)
somc –up myclass1        (On Windows)
```

This command will place all of the information in the "myclass1.idl" file, including the private portions, in the Interface Repository.

If you are using tools that understand SOM and rely on the Interface Repository to describe the types and instance data in your classes, you may need to include the private sections from your IDL source files when building the Interface Repository.

7.4 Programming with the Interface Repository Objects

The SOM Interface Repository Framework provides an object-oriented programming interface to the IDL information processed by the SOM Compiler. Unlike many frameworks that require you to inherit their behavior in order to use it, the Interface Repository Framework is useful in its own right as a set of predefined objects that you can access to obtain information. Of course, if you need to subclass a class to modify its behavior, you can certainly do so; but typically this is not necessary.

The SOM Interface Repository contains the fully-analyzed (compiled) contents of all information in an IDL source file. This information takes the the form of persistent objects that can be accessed from a running program. There are ten classes of objects in the Interface Repository that correspond directly to the major elements in IDL source files; in addition, one instance of another class exists outside of the IR itself, as follows:

Contained — All objects in the Interface Repository are instances of classes derived from this class and exhibit the common behavior defined in this interface.

Container — Some objects in the Interface Repository hold (or contain) other objects. (For example, a module [**ModuleDef**] can contain an interface [**InterfaceDef**].) All Interface Repository objects that hold other objects are instances of classes derived from this class and exhibit the common behavior defined by this class.

ModuleDef — An instance of this class exists for each **module** defined in an IDL source file. **ModuleDef**s are **Container**s, and they can hold **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **InterfaceDef**s, and other **ModuleDef**s.

InterfaceDef — An instance of this class exists for each **interface** named in an IDL source file. (One **InterfaceDef** corresponds to one SOM class.) **InterfaceDef**s are **Container**s, and they can hold **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **AttributeDef**s, and **OperationDef**s.

AttributeDef — An instance of this class exists for each **attribute** defined in an IDL source file. **AttributeDef**s are found only inside of (contained by) **InterfaceDef**s.

OperationDef — An instance of this class exists for each **operation** (method) defined in an IDL source file. **OperationDef**s are **Container**s that can hold **ParameterDef**s. **OperationDef**s are found only inside of (contained by) **InterfaceDef**s.

ParameterDef — An instance of this class exists for each **parameter** of each operation (method) defined in an IDL source file. **ParameterDef**s are found only inside of (contained by) **OperationDef**s.

TypeDef — An instance of this class exists for each **typedef**, **struct**, **union**, or **enum** defined in an IDL source file. **TypeDef**s may be found inside of (contained by) any Interface Repository **Container** except an **OperationDef**.

ConstantDef — An instance of this class exists for each **constant** defined in an IDL source file. **ConstantDef**s may be found inside (contained by) of any Interface Repository **Container** except an **OperationDef**.

ExceptionDef — An instance of this class exists for each **exception** defined in an IDL source file. **ExceptionDef**s may be found inside of (contained by) any Interface Repository **Container** except an **OperationDef**.

Repository — One instance of this class exists for the entire SOM Interface Repository, to hold IDL elements that are global in scope. The instance of this class does not, however, reside within the IR itself.

Methods introduced by Interface Repository classes

The Interface Repository classes introduce nine new methods, which are briefly described below. Many of the classes simply override methods to customize them for the corresponding IDL element; this is particularly true for classes representing IDL elements that are only contained within another syntactic element. Full descriptions of each method are found in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

• **Contained** class methods (*all* IR objects are instances of this class and exhibit this behavior):

describe — Returns a structure of type **Description** containing all information defined in the IDL specification of the syntactic element corresponding to the target **Contained** object. For example, for a target **InterfaceDef** object, the **describe** method returns information about the IDL interface declaration. The **Description** structure contains a "name" field with an identifier that categorizes the description (such as, "InterfaceDescription") and a "value" field holding an "any" structure that points to another structure containing the IDL information for that particular element (in this example, the interface's IDL specifications).

within — Returns a sequence designating the object(s) of the IR within which the target **Contained** object is contained. For example, for a target **TypeDef** object, it might be contained within any other IR object(s) except an **OperationDef** object.

• **Container** class methods (*some* IR objects contain other objects and exhibit this behavior):

contents — Returns a sequence of pointers to the object(s) of the IR that the target **Container** object contains. (For example, for a target **InterfaceDef** object, the **contents** method returns a pointer to each IR object that corresponds to a part of the IDL interface declaration.) The method provides options for excluding inherited objects or for limiting the search to only a specified kind of object (such as **AttributeDef**s).

describe_contents — Combines the **describe** and **contents** methods; returns a sequence of **ContainerDescription** structures, one for each object contained by the target **Container** object. Each structure has a pointer to the related object, as well as "name" and "value" fields resulting from the **describe** method.

lookup_name — Returns a sequence of pointers to objects of a given name contained within a specified **Container** object, or within (sub)objects contained in the specified **Container** object.

- **ModuleDef** class methods:

  — Override **describe** and **within.**

- **InterfaceDef** class methods:

  describe_interface — Returns a description of all methods and attributes of a given interface definition object that are held in the Interface Repository.

  — Also overrides **describe** and **within.**

- **AttributeDef** class method:

  — Overrides **describe.**

- **OperationDef** class method:

  — Overrides **describe.**

- **ParameterDef** class method:

  — Overrides **describe.**

- **TypeDef** class method:

  — Overrides **describe.**

- **ConstantDef** class method:

  — Overrides **describe.**

- **ExceptionDef** class method:

  — Overrides **describe.**

- **Repository** class methods:

  lookup_id — Returns the **Contained** object that has a specified **RepositoryId.**

  lookup_modifier — Returns the string value held by a SOM or user-defined **modifier**, given the name and type of the modifier, and the name of the object that contains the **modifier.**

  release_cache — Releases, from the internal object cache, the storage used by all currently unreferenced Interface Repository objects.

Accessing objects in the Interface Repository

As mentioned above, one instance of the **Repository** class exists for the entire SOM Interface Repository. This object does not, itself, reside in the Interface Repository (hence it does not exhibit any of the behavior defined by the **Contained** class). It is, however, a **Container**, and it holds all **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **InterfaceDef**s, and **ModuleDef**s that are global in scope (that is, not contained inside of any other **Container**s).

When any method provided by the **Repository** class is used to locate other objects in the Interface Repository, those objects are automatically instantiated and activated. Consequently, when the program is finished using an object from the Interface Repository, the client code should release the object using the **somFree** method.

All objects contained in the Interface Repository have both a “name” and a “Repository ID” associated with them. The name is not guaranteed to be unique, but it does uniquely identify an object within the context of the object that contains it. The Repository ID of each object is guaranteed to uniquely identify that object, regardless of its context.

For example, two **TypeDef** objects may have the same name, provided they occur in separate name scopes (**ModuleDef**s or **InterfaceDef**s). In this case, asking the Interface Repository to locate the **TypeDef** object based on its name would result in both **TypeDef** objects being returned. On the other hand, if the name is looked up from a particular **ModuleDef** or **InterfaceDef** object, only the **TypeDef** object within the scope of that **ModuleDef** or **InterfaceDef** would be returned. By contrast, once the Repository ID of an object is known, that object can always be directly obtained from the **Repository** object via its Repository ID.

C or C++ programmers can obtain an instance of the **Repository** class using the **RepositoryNew** macro. Programmers using other languages (and C/C++ programmers without static linkage to the **Repository** class) should invoke the method **somGetInterfaceRepository** on the **SOMClassMgrObject**. For example,

For C or C++ (static linkage):

```
#include <repostry.h>
Repository repo;

...

repo = RepositoryNew();
```

From other languages (and for dynamic linkage in C/C++):

1. Use the **somEnvironmentNew** function to obtain a pointer to the **SOMClassMgrObject**, as described in Chapter 3, "Using SOM Classes in Client Programs."

2. Use the **somResolve** or **somResolveByName** function to obtain a pointer to the **somGetInterfaceRepository** method procedure.

3. Invoke the method procedure on the **SOMClassMgrObject**, with no additional arguments, to obtain a pointer to the **Repository** object.

After obtaining a pointer to the **Repository** object, use the methods it inherits from **Container** or its own **lookup_id** method to instantiate objects in the Interface Repository. As an example, the **contents** method shown in the C fragment below activates every object with global scope in the Interface Repository and returns a sequence containing a pointer to every global object:

```
#include <containd.h>      /* Behavior common to all IR objects */
Environment *ev;
int i;
sequence(Contained) everyGlobalObject;

ev = SOM_CreateLocalEnvironment(); /* Get an environment to use */
printf ("Every global object in the Interface Repository:\n");

everyGlobalObject = Container_contents (repo, ev, "all", TRUE);

for (i=0; i < everyGlobalObject._length; i++) {
    Contained aContained;

    aContained = (Contained) everyGlobalObject._buffer[i];
    printf ("Name: %s, Id: %s\n",
        Contained__get_name (aContained, ev),
        Contained__get_id (aContained, ev));
    SOMObject_somFree (aContained);
}
```

Taking this example one step further, here is a complete program that accesses every object in the entire Interface Repository. It, too, uses the **contents** method, but this time recursively calls the **contents** method until every object in every container has been found:

```
#include <stdio.h>
#include <containd.h>
#include <repostry.h>

void showContainer (Container c, int *next);

main ()
{
    int count = 0;
    Repository repo;

    repo = RepositoryNew ();
    printf (”Every object in the Interface Repository:\n\n”);
    showContainer ((Container) repo, &count);
    SOMObject_somFree (repo);
    printf (”%d objects found\n”, count);
    exit (0);
}

void showContainer (Container c, int *next)
{
    Environment *ev;
    int i;
    sequence(Contained) everyObject;

    ev = SOM_CreateLocalEnvironment (); /* Get an environment */
    everyObject = Container_contents (c, ev, ”all”, TRUE);

    for (i=0; i<everyObject._length; i++) {
        Contained aContained;

        (*next)++;
        aContained = (Contained) everyObject._buffer[i];
        printf (”%6d. Type: %-12s id: %s\n”, *next,
            SOMObject_somGetClassName (aContained),
            Contained__get_id (aContained, ev));
        if (SOMObject_somIsA (aContained, _Container))
            showContainer ((Container) aContained, next);
        SOMObject_somFree (aContained);
    }
}
```

Once an object has been retrieved, the methods and attributes appropriate for that particular object can then be used to access the information contained in the object. The methods supported by each class of object in the Interface Repository, as well as the classes themselves, are documented in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

A word about memory management

Several conventions are built into the SOM Interface Repository with regard to memory management. You will need to understand these conventions to know when it is safe and appropriate to free memory references and also when it is your responsibility to do so.

All methods that access attributes (such as, the `_get_<attribute>` methods) always return either simple values or direct references to data within the target object. This is necessary because these methods are heavily used and must be fast and efficient. Consequently, you should never free any of the memory references obtained through attributes. This memory will be released automatically when the object that contains it is freed.

For all methods that give out object references (there are five: **within, contents, lookup_name, lookup_id,** and **describe_contents**), when finished with the object, you are expected to release the object reference by invoking the **somFree** method. (This is illustrated in the sample program that accesses all Interface Repository objects.) Do not release the object reference until you have either copied or finished using all of the information obtained from the object.

The **describe** methods (**describe, describe_contents,** and **describe_interface**) return structures and sequences that contain information. The actual structures returned by these methods are passed by value (and hence should only be freed if you have allocated the memory used to receive them). However, you may be required to free some of the information contained in the returned structures when you are finished. Consult the specific method in the *SOMobjects Developer Toolkit: Programmers Reference Manual* for more details about what to free.

During execution of the **describe** and **lookup** methods, sometimes intermediate objects are activated automatically. These objects are kept in an internal cache of objects that are in use, but for which no explicit object references have been returned as results. Consequently, there is no way to identify or free these objects individually. However, whenever your program is finished using all of the information obtained thus far from the Interface Repository, invoking the **release_cache** method causes the Interface Repository to purge its internal cache of these implicitly referenced objects. This cache will replenish itself automatically if the need to do so subsequently arises.

Using TypeCode pseudo-objects

Much of the detailed information contained in Interface Repository objects is represented in the form of **TypeCode**s. **TypeCode**s are complex data structures whose actual representation is hidden. A **TypeCode** is an architected way of describing in complete detail everything that is known about a particular data type in the IDL language, regardless of whether it is a (built-in) *basic* type or a (user-defined) *aggregate* type.

Conceptually, every **TypeCode** contains a “kind” field (which classifies it), and one or more parameters that carry descriptive information appropriate for that particular category of **TypeCode**. For example, if the data type is **long**, its **TypeCode** would contain a “kind” field with the value **tk_long**. No additional parameters are needed to completely describe this particular data type, since **long** is a basic type in the IDL language.

By contrast, if the **TypeCode** describes an IDL **struct**, its “kind” field would contain the value **tk_struct**, and it would possess the following parameters: a string giving the name of the struct, and two additional parameters for each member of the struct: a string giving the member name and another (inner) **TypeCode** representing the member’s type. This example illustrates the fact that **TypeCode**s can be nested and arbitrarily complex, as appropriate to express the type of data they describe. Thus, a structure that has N members will have a **TypeCode** of **tk_struct** with 2N+1 parameters (a name and **TypeCode** parameter for each member, plus a name for the struct itself).

A **tk_union TypeCode** representing a union with N members has 3N+2 parameters: the type name of the union, the **switch TypeCode**, and a label value, member name and associated **TypeCode** for each member. (The label values all have the same type as the switch, except that the default member, if present, has a label value of zero **octet.)**

A **tk_enum TypeCode** (which represents an enum) has N+1 parameters: the name of the enum followed by a string for each enumeration identifier. A **tk_string TypeCode** has a single parameter: the maximum string length, as an integer. (A maximum length of zero signifies an unbounded string.)

A **tk_sequence TypeCode** has two parameters: a **TypeCode** for the sequence elements, and the maximum size, as an integer. (Again, zero signifies unbounded.)

A **tk_array TypeCode** has two parameters: a **TypeCode** for the array elements, and the array length, as an integer. (Arrays must be bounded.)

The **tk_objref TypeCode** represents an object reference; its parameter is a repository ID that identifies its interface.

A complete table showing the parameters of all possible **TypeCodes** is given in the *SOMobjects Developer Toolkit Programmers Reference Manual;* see the **TypeCode_kind** function of the Interface Repository Framework.

TypeCodes are not actually "objects" in the formal sense. **TypeCode**s are referred to in the CORBA standard as *pseudo-objects* and described as "opaque". This means that, in reality, **TypeCode**s are special data structures whose precise definition is not fully exposed. Their implementation can vary from one platform to another, but all implementations must exhibit a minimal set of architected behavior. SOM **TypeCode**s support the architected behavior and have additional capability as well (for example, they can be copied and freed).

Although **TypeCode**s are not objects, the programming interfaces that support them adhere to the same conventions used for IDL method invocations in SOM. That is, the first argument is always a **TypeCode** pseudo-object, and the second argument is a pointer to an **Environment** structure. Similarly, the names of the **TypeCode** functions are constructed like SOM's C-language method-invocation macros (all functions that operate on **TypeCode**s are named **TypeCode_**<*function–name*>). Because of this ostensible similarity to an IDL class, the **TypeCode** programming interfaces can be conveniently defined in IDL as shown below.

```
interface TypeCode {

enum TCKind {
    tk_null, tk_void,
    tk_short, tk_long, tk_ushort, tk_ulong,
    tk_float, tk_double, tk_boolean, tk_char,
    tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
    tk_struct, tk_union, tk_enum, tk_string,
    tk_sequence, tk_array,

    // The remaining enumerators are SOM-unique extensions
    // to the CORBA standard.
    //
    tk_pointer, tk_self, tk_foreign
};

exception Bounds {};
// This exception is returned if an attempt is made
// by the parameter() operation (described below) to
// access more parameters than exist in the receiving
// TypeCode.

boolean equal (in TypeCode tc);
// Compares the argument with the receiver and returns TRUE
// if both TypeCodes are equivalent.  This is NOT a test for
// identity.

TCKind kind ();
// Returns the type of the receiver as a TCKind.

long param_count ();
// Returns the number of parameters that make up the
// receiving TypeCode.

any parameter (in long index) raises (Bounds);
// Returns the indexed parameter from the receiving TypeCode.
// Parameters are indexed from 0 to param_count()-1.
```

```
//
//  The remaining operations are SOM-unique extensions.
//

short alignment ();
// This operation returns the alignment required for an instance
// of the type described by the receiving TypeCode.

TypeCode copy (in TypeCode tc);
// This operation returns a copy of the receiving TypeCode.

void free (in TypeCode tc);
// This operation frees the memory associated with the
// receiving TypeCode.  Subsequently, no further use can be
// made of the receiver, which, in effect, ceases to exist.

void print (in TypeCode tc);
// This operation writes a readable representation of the
// receiving TypeCode to stdout.  Useful for examining
// TypeCodes when debugging.

void setAlignment (in short align);
// This operation overrides the required alignment for an
// instance of the type described by the receiving TypeCode.

long size (in TypeCode tc);
// This operation returns the size of an instance of the
// type represented by the receiving TypeCode.
};
```

A detailed description of the programming interfaces for **TypeCode**s is given in the *SOMobjects Developer Toolkit: Programmers Reference Manual*.

Providing 'alignment' information

In addition to the parameters in the **TypeCode**s that describe each type, a SOM-unique extension to the **TypeCode** functionality allows each **TypeCode** to carry alignment information as a "hidden" parameter. Use the **TypeCode_alignment** function to access the alignment value. The alignment value is a short integer that should evenly divide any memory address where an instance of the type will occur.

If no alignment information is provided in your IDL source files, all **TypeCode**s carry default alignment information. The default alignment for a type is the natural boundary for the type, based on the natural boundary for the basic types of which it may be composed. This information can vary from one hardware platform to another. The **TypeCode** will contain the default alignment information appropriate to the platform where it was defined.

To provide alignment information for the types and instances of types in your IDL source file, use the "align=N" modifier, where N is your specified alignment. Use standard modifier syntax of the SOM Compiler to attach the alignment information to a particular element in the IDL source file. In the following example, `align=1` (that is, unaligned or no alignment) is attached to the struct "abc" and to one particular instance of struct "def" (the instance data item "y").

```
interface i {
    struct abc {
        long a;
        char b;
        long c;
    };
    struct def {
        char l;
        long m;
    };
    void foo ();
    implementation {
    //# instance data
        abc x;
        def y;
        def z;


    //# alignment modifiers
        abc: align=1;
        y: align=1;
    };
};
```

Be aware that assigning the required alignment information to a type does *not* guarantee that instances of that type will actually be aligned as indicated. To ensure that, you must find a way to instruct your compiler to provide the desired alignment. In practice, this can be difficult except in simple cases. Most compilers can be instructed to treat all data as aligned (that is, default alignment) or as unaligned, by using a compile-time option or #pragma. The more important consideration is to make certain that the **TypeCode**s going into the Interface Repository actually reflect the alignment that your compiler provides. This way, when programs (such as the DSOM Framework) need to interpret the layout of data during their execution, they will be able to accurately map your data structures. This happens automatically when using the normal default alignment.

If you wish to use unaligned instance data when implementing a class, place an "unattached" `align=1` modifier in the implementation section. An unattached `align=N` modifier is presumed to pertain to the class's instance data structure, and will by implication be attached to all of the instance data items.

When designing your own public types, be aware that the best practice of all (and the one that offers the best opportunity for language neutrality) is to lay out your types carefully so that it will make no difference whether they are compiled as aligned or unaligned!

Using the 'tk_foreign' TypeCode

TypeCodes can be used to partially describe types that cannot be described in IDL (for example, a FILE type in C, or a specific class type in C++). The SOM-unique extension **tk_foreign** is used for this purpose. A **tk_foreign TypeCode** contains three parameters:

1. The name of the type,
2. An implementation context string, and
3. A length.

The implementation context string can be used to carry an arbitrarily long description that identifies the context where the foreign type can be used and understood. If the length of the type is also known, it can be provided with the length parameter. If the length is not known or is not constant, it should be specified as zero. If the length is not specified, it will default to the size of a pointer. A **tk_foreign TypeCode** can also have alignment information specified, just like any other **TypeCode**.

Using the following steps causes the SOM Compiler to create a foreign **TypeCode** in the Interface Repository:

1. Define the foreign type as a **typedef** SOMFOREIGN in the IDL source file.
2. Use the **#pragma modifier** statement to supply the additional information for the **TypeCode** as modifiers. The implementation context information is supplied using the "impctx" modifier.
3. Compile the IDL file using the **–u** option to place the information in the Interface Repository.

For example:

```
typedef SOMFOREIGN Point;
#pragma modifier Point: impctx="C++ Point class",length=12,align=4;
```

If a foreign type is used to define instance data, structs, unions, attributes, or methods in an IDL source file, it is your responsibility to ensure that the implementation and/or usage bindings contain an appropriate definition of the type that will satisfy your compiler. You can use the **passthru** statement in your IDL file to supply this definition. However, it is *not* recommended that you expose foreign data in attributes, methods, or any of the public types, if this can be avoided, because there is no guarantee that appropriate usage binding information can be provided for all languages. If you know that all users of the class will be using the same implementation language that your class uses, you may be able to disregard this recommendation.

TypeCode constants

TypeCodes are actually available in two forms: In addition to the **TypeCode** information provided by the methods of the Interface Repository, **TypeCode** constants can be generated by the SOM Compiler in your C or C++ usage bindings upon request. A **TypeCode** constant contains the same information found in the corresponding IR **TypeCode**, but has the advantage that it can be used as a literal in a C or C++ program anywhere a normal **TypeCode** would be acceptable.

TypeCode constants have the form **TC_**<*typename*>, where <*typename*> is the name of a type (that is, a typedef, union, struct, or enum) that you have defined in an IDL source file. In addition, all IDL basic types and certain types dictated by the OMG CORBA standard come with pre-defined **TypeCode** constants (such as **TC_long**, **TC_short**, **TC_char**, and so forth). A full list of the pre-defined **TypeCode** constants can be found in the file "somtcnst.h". You must explicitly include this file in your source program to use the pre-defined **TypeCode** constants.

Since the generation of **TypeCode** constants can increase the time required by the SOM Compiler to process your IDL files, you must explicitly request the production of **TypeCode** constants if you need them. To do so, use the "tcconsts" modifier with the **–m** option of the **sc** or **somc** command. For example, the command

```
sc –sh –mtcconsts myclass.idl           (On AIX or Windows)
somc –sh –mtcconsts myclass.idl         (On Windows)
```

will cause the SOM Compiler to generate a "myclass.h" file that contains **TypeCode** constants for the types defined in "myclass.idl".

Using the IDL basic type 'any'

Some Interface Repository methods and **TypeCode** functions return information typed as the IDL basic type **any**. Usually this is done when a wide variety of different types of data may need to be returned through a common interface. The type **any** actually consists of a structure with two fields: a **_type** field and a **_value** field. The **_value** field is a pointer to the actual datum that was returned, while the **_type** field holds a **TypeCode** that describes the datum.

In many cases, the context in which an operation occurs makes the type of the datum apparent. If so, there is no need to examine the **TypeCode** unless it is simply as a consistency check. For example, when accessing the first parameter of a **tk_struct TypeCode**, the type of the result will always be the name of the structure (a string). Because this is known ahead of time, there is no need to examine the returned **TypeCode** in the **any _type** field to verify that it is a **tk_string TypeCode**. You can just rely on the fact that it is a string; or, you can check the **TypeCode** in the **_type** field to verify it, if you so choose.

An IDL **any** type can be used in an interface as a way of bypassing the strong type checking that occurs in languages like ANSI C and C++. Your compiler can only check that the interface returns the **any** structure; it has no way of knowing what type of data will be carried by the **any** during execution of the program. Consequently, in order to write C or C++ code that accesses the contents of the **any** correctly, you must always cast the **_value** field to reflect the actual type of the datum at the time of the access.

Here is an example of a code fragment written in C that illustrates how the casting must be done to extract various values from an **any**:

```
#include <som.h>    /* For "any" & "Environment" typedefs */
#include <somtc.h>  /* For TypeCode_kind prototype        */

any result;
Environment *ev;

printf ("result._value = ");
switch (TypeCode_kind (result._type, ev)) {

    case tk_string:
        printf ("%s\n", *((string *) result._value));
        break;

    case tk_long:
        printf ("%ld\n", *((long *) result._value));
        break;

    case tk_boolean:
        printf ("%d\n", *((boolean *) result._value));
        break;

    case tk_float:
        printf ("%f\n", *((float *) result._value));
        break;

    case tk_double:
        printf ("%f\n", *((double *) result._value));
        break;

    default:
        printf ("something else!\n");
}
```

Note: Of course, an **any** has no restriction, per se, on the type of datum that it can carry. Frequently, however, methods that return an **any** or that accept an **any** as an argument do place semantic restrictions on the actual type of data they can accept or return. Always consult the reference page for a method that uses an **any** to determine whether it limits the range of types that may be acceptable.

Chapter 8. The Persistence Framework

Contents

Chapter 8. The Persistence Framework

8.1 Introduction

This chapter and the programs it describes demonstrate the use of the Persistence Framework of the SOMobjects Toolkit. It is assumed that the reader is proficient in the C programming language and has a working knowledge of Object-Oriented Programming (OOP) with SOM.

The SOMobjects Persistence Framework allows SOM objects to be saved in a persistent state and then later restored. The term “persistent” as used here means that an object’s state can be preserved beyond the termination of the process that creates it. Objects can be stored by themselves or grouped with other objects. Objects can be stored in default formats or in specially designed formats. Objects can be stored in the file system or in more specialized repositories. The examples in this chapter assume that objects are being stored in the file system by the default classes of the framework.

Functionally, the Persistence Framework replaces the file-system interface of a programming language. The Persistence Framework is not, however, designed to compete with a database management system. It does not provide recovery, transactions, or multi-user record locking. These should be handled at a higher level.

Several of the classes of the Persistence Framework can be subclassed to customize how and where objects are stored. This chapter deals primarily with the default framework classes. For a detailed discussion of subclassing the framework to change its default behavior, refer to Appendix D, “Subclassing the Persistence Framework.”

The next section describes a telephone-directory application, which will be used throughout this chapter to illustrate various features of the Persistence Framework. Subsequent sections discuss different aspects of persistent objects, techniques for saving and restoring persistent objects, managing persistent objects, and lastly, advanced topics on customizing the Persistence Framework.

Release 2.1 note: Many of the examples in this chapter make use of the **somInit** and **somUninit** methods. Although these methods have been superseded by the **somDefaultInit** and **somDestruct** methods, which are more efficient, be assured that **somInit** still executes correctly. When developing your own applications, however, you may wish to override **somDefaultInit** instead of **somInit** to customize object initialization.

8.2 The Telephone-Directory Application

This chapter uses example programs to demonstrate using the SOMobjects Persistence Framework. Each example is a variation on a telephone-directory application. The telephone-directory application creates a list of phone numbers, stores the list, recalls the list, and displays its contents. This section describes the basic, nonpersistent version of the telephone application. Later examples use the Persistence Framework to store and restore the list.

The telephone-directory application includes SOM classes: A directory-entry class (“dirEntry”) and a phone-directory class (“phoneDir”). An object of class “phoneDir” encapsulates the phone directory so that individual entires can be added to or deleted from the phone directory. The “phoneDir” class is defined as follows:

Example 1: Nonpersistent telephone-directory example

```
#include <somobj.idl>
interface dirEntry;

interface phoneDir : SOMObject{
   const unsigned long MAXDIRSIZE = 16;
   attribute sequence<dirEntry, MAXDIRSIZE> directory;

   long addEntry(in dirEntry entry);
// Adds a dirEntry object to the directory.

   void printDirInfo();
// Invokes lsEntry for each instance of direntry.

   dirEntry getEntry(in string entryID);
// getEntry returns a pointer to the list entry with name == entryID.
// If a matching entry is not found, it returns NULL.

   dirEntry dropEntry(in string entryID);
// dropEntry removes the entry with name == entryID from the directory
// array, and returns a pointer to the dropped entry.
// If a matching entry is not found, it returns NULL.
// The client application must use sompDeleteObject on the returned object
// to delete it from the IO Group.  The client must also free the dirEntry
// Object returned.

   #ifdef __SOMIDL__

   implementation
   {
      callstyle=oidl;
      releaseorder: addEntry, printDirInfo, getEntry, dropEntry,
                    _get_directory, _set_directory;

   // Class Modifiers
      filestem = phonedir;

      passthru C_h =  ”#include <direntry.h>”;

   // Method Modifiers
      somInit: override;
      somUninit: override;
    };
   #endif /* __SOMIDL__ */
};
```

The implementation of “phoneDir” follows:

```
#define phoneDir_Class_Source
#define SOM_Module_phonedir_Source
#include <phonedir.ih>
```

```
SOM_Scope long  SOMLINK addEntry(phoneDir somSelf, dirEntry entry)
{
    Environment *ev;
    long rc;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","addEntry");

    ev = SOM_CreateLocalEnvironment();
    if (sequenceLength(_directory) < sequenceMaximum(_directory)) {
        sequenceElement(_directory, sequenceLength(_directory)) = entry;
        sequenceLength(_directory)++;
        rc = 0L;
    } else
        rc = -1L;
    SOM_DestroyLocalEnvironment(ev);
    return(rc);
}

SOM_Scope void  SOMLINK printDirInfo(phoneDir somSelf)
{
    int probe;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","printDirInfo");

    if (sequenceLength(_directory) > 0) {
       somPrintf("\n");
       somPrintf("Name                Phone\n");
       somPrintf("----------------    ----------------\n");
       for (probe = 0; probe < sequenceLength(_directory); probe++) {
          _lsEntry(sequenceElement(_directory, probe));
       }
    } else {
        somPrintf ("\nDirectory is Empty\n");
    }
}

SOM_Scope dirEntry  SOMLINK getEntry(phoneDir somSelf, string entryID)
{
    int probe;
    char *probename;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","getEntry");

    for (probe = 0;probe < sequenceLength(_directory);probe++) {
      probename = dirEntry__get_name(sequenceElement(_directory,
                                                     probe));
      if (strcmp(probename, entryID) == 0)
          return (sequenceElement(_directory, probe));
    } /* endfor */

    return (dirEntry) NULL;
}
```

```
SOM_Scope dirEntry  SOMLINK dropEntry(phoneDir somSelf, string entryID)
{
    int probe;
    char *probename;
    dirEntry dptr;
    Environment *ev;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","dropEntry");

    ev = SOM_CreateLocalEnvironment();
    for (probe = 0;probe < sequenceLength(_directory);probe++) {
        probename = dirEntry__get_name(sequenceElement(_directory,
                                                       probe));
        if (strcmp(probename, entryID) == 0) {
            dptr = sequenceElement(_directory, probe);
            sequenceLength(_directory)--;
            for (; probe < sequenceLength(_directory); probe++) {
                sequenceElement(_directory, probe) =
                sequenceElement(_directory, (probe + 1));
            }
            SOM_DestroyLocalEnvironment(ev);
            return dptr;
        } /* endif */
    } /* endfor */

    SOM_DestroyLocalEnvironment(ev);
    return (dirEntry) NULL;
}

SOM_Scope void  SOMLINK somInit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somInit");
    phoneDir_parent_SOMObject_somInit(somSelf);

    sequenceMaximum(_directory) = MAXDIRSIZE;
    sequenceLength(_directory) = 0;
    _directory._buffer = SOMMalloc(sizeof (dirEntry) * MAXDIRSIZE);

}

SOM_Scope void  SOMLINK somUninit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somUninit");
    phoneDir_parent_SOMObject_somUninit(somSelf);

    if (_directory._buffer) SOMFree(_directory._buffer);
}
```

A “dirEntry” object encapsulates the name and phone number of a single entry in the telephone directory. Methods are provided for assigning name and phone number values to an object, and for displaying the object’s contents. The “dirEntry” is defined as shown here:

```
#include <somobj.idl>
interface dirEntry : SOMObject
{
   attribute string name;
   attribute string phone;

   void mkEntry(in string name, in string phone_no);
// Initialize new entry.

   void lsEntry();
// lsEntry displays the entry on a single line.

   #ifdef __SOMIDL__
   implementation
   {
      callstyle=oidl;
      // no Environment on methods

      releaseorder: mkEntry, lsEntry,
                    _get_name, _set_name,
                    _get_phone, _set_phone;

      // Class modifiers
      filestem = direntry;

      // Method modifiers
      somInit: override;
      somUninit: override;

   };
   #endif /* __SOMIDL__ */
};
```

The following code gives the implementation of “dirEntry”:

```
#ifndef SOM_Module_direntry_Source
#define SOM_Module_direntry_Source
#endif
#define dirEntry_Class_Source
#include <direntry.ih>
#include <string.h>
SOM_Scope void  SOMLINK mkEntry(dirEntry somSelf, string name,
                                string phone_no)
{
    dirEntryData *somThis = dirEntryGetData(somSelf);
    dirEntryMethodDebug(”dirEntry”,”mkEntry”);

    if (_name) SOMFree(_name);
    _name = (string)SOMMalloc(strlen(name)+1);
    strcpy (_name, name);
    if (_phone) SOMFree(_phone);
    _phone = (string)SOMMalloc(strlen(phone_no)+1);
    strcpy (_phone, phone_no);
}
```

```
SOM_Scope void  SOMLINK lsEntry(dirEntry somSelf)
{
    dirEntryData *somThis = dirEntryGetData(somSelf);
    dirEntryMethodDebug("dirEntry","lsEntry");

    somPrintf ("%-16s    %-16s\n", _name, _phone);
}


SOM_Scope void  SOMLINK somInit(dirEntry somSelf)
{
    dirEntryData *somThis = dirEntryGetData(somSelf);
    dirEntryMethodDebug("dirEntry","somInit");
    dirEntry_parent_SOMObject_somInit(somSelf);

    _name = NULL;
    _phone = NULL;
}
SOM_Scope void  SOMLINK somUninit(dirEntry somSelf)
{
    dirEntryData *somThis = dirEntryGetData(somSelf);
    dirEntryMethodDebug("dirEntry","somUninit");
    dirEntry_parent_SOMObject_somUninit(somSelf);

    if (_name) SOMFree(_name);
    if (_phone) SOMFree(_phone);
}
```

The following test program demonstrates the use of the “phoneDir” class:

```
#include "phonedir.h"        /*  Client Class Includes */
#include "direntry.h"


main()
{
    long int rc;

    dirEntry  name1, name2;
    phoneDir  mylist;
    Environment *ev;

/*  Initialize system.
    ------------------ */
    ev = SOM_CreateLocalEnvironment();



/*  Create a directory with two entries.
    ------------------------------------ */
    mylist = phoneDirNew();
    name1 = dirEntryNew();
    _mkEntry  (name1, ev, "Roger", "555-8585");
    _addEntry (mylist, ev, name1);

    name2 = dirEntryNew();
    _mkEntry  (name2, ev, "Charles", "555-1717");
    _addEntry (mylist, ev, name2);
```

```
/*  Display the directory.
    ----------------------*/
    _printDirInfo(mylist, ev);

/*  Shutdown.
    --------- */
    SOM_DestroyLocalEnvironment(ev);
}
```

This program produces the following output:

```
Name                Phone
----------------    ----------------
Roger               555-8585
Charles             555-1717
```

8.3 Persistent Objects

A persistent object is one whose state can be preserved beyond the termination of the process that creates it. An object is potentially persistent if its class is derived (either directly or indirectly) from the **SOMPPersistentObject** class, either through single or multiple inheritance. If through single inheritance, the class of persistent objects (such as “phoneDir”) is usually derived directly from **SOMPPersistentObject**. If through multiple inheritance, typically a nonpersistent version of the class already exists, and then the persistent version of the class is derived from both the nonpersistent version and **SOMPPersistentObject.**

The following definition of the “phoneDir” class from the telephone directory application demonstrates the single inheritance case. The differences between the nonpersistent and the persistent versions of “phoneDir” are shown in bold.

Example 2: Single inheritance definition of persistent “phoneDir”

```
#include <po.idl>

interface dirEntry;

interface phoneDir : SOMPPersistentObject
{
   const unsigned long MAXDIRSIZE = 16;
   attribute sequence<dirEntry, MAXDIRSIZE> directory;

   long addEntry(in dirEntry entry);
// Adds a dirEntry object to the directory.

   void printDirInfo();
// Invokes lsEntry for each instance of direntry.

   dirEntry getEntry(in string entryID);
// getEntry returns a pointer to the list entry with name == entryID.
// If a matching entry is not found, it returns NULL.

   dirEntry dropEntry(in string entryID);
// dropEntry removes the entry with name == entryID from the directory
// array, and returns a pointer to the dropped entry.
// If a matching entry is not found, it returns NULL.
// The client application must use sompDeleteObject on the returned object
// to delete it from the IO Group.  The client must also free the dirEntry
// Object returned.

   #ifdef __SOMIDL__

   implementation
   {
      callstyle=oidl;

      releaseorder: addEntry, printDirInfo, getEntry, dropEntry,
                    _get_directory, _set_directory;

   // Class Modifiers
      filestem = phonedir;

      passthru C_h =  ”#include <direntry.h>”;

   // Attribute Modifiers
      directory: persistent;

   // Method Modifiers
      somInit: override;
      somUninit: override;
      sompIsDirty: override;
   };
   #endif /* __SOMIDL__ */

};
```

To make the object persistent we have derived it from **SOMPPersistentObject**. The **SOMPPersistentObject** class is defined in the po.idl file which is included at the beginning of the file.

Observe that Example 2 includes the following new statement, which registers the “directory” attribute as a persistent attribute:

```
directory: persistent;
```

In order to use the default format (class **SOMPAttrEncoderDecoder**) for writing persistent object data to a file, persistent object classes must fulfill the following design criteria:

- All persistent data must be one of these standard CORBA types:
  short, long, unsigned short, unsigned long, float, double, boolean, char, octet, string, sequence, structure, union, enum, array, union, or a persistent object.
- All persistent data must be declared as an **attribute**, and each of these attributes must be signified by the attribute modifier **persistent**. The **persistent** modifier for an attribute may appear in a derived class.
- Your persistent object interface definitions and those which they are derived from, if they contain persistent data, must be put in the SOM Interface Repository. This is done by compiling your .idl files with the –u flag, for example:

```
sc –sir –u myclass.idl          (On AIX or OS/2),
somc –sir –u myclass.idl        (On Windows)
```

Finally, observe that the **SOMPPersistentObject** method **sompIsDirty** has been overridden. The **sompIsDirty** method is responsible for reporting whether the object is “dirty”. That is, that the object’s persistent data has been changed. By default, the **sompIsDirty** method always returns TRUE.

The implementation of the persistent “phoneDir” class is, for the most part, unchanged from the nonpersistent “phoneDir” class, except for the addition of the lines shown in bold in the following methods and the addition of the **sompIsDirty** implementation. Also take note of the additions to **somInit** and **somUninit**. These methods must pass through to their parent implementations for the Persistence Framework to initialize your object correctly.

Implementation of the persistent telephone directory

The following highlights the modifications to the “addEntry” method:

```
SOM_Scope long  SOMLINK addEntry(phoneDir somSelf, dirEntry entry)
{
    Environment *ev;
    long rc;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug(”phoneDir”,”addEntry”);

    ev = SOM_CreateLocalEnvironment();
    if (sequenceLength(_directory) < sequenceMaximum(_directory)) {
        sequenceElement(_directory, sequenceLength(_directory)) = entry;
        sequenceLength(_directory)++;
        _sompSetDirty (somSelf, ev);
        rc = 0L;
    } else
        rc = –1L;
    SOM_DestroyLocalEnvironment(ev);
    return(rc);
}
```

The modification of the “dropEntry” method is similar to the “addEntry” method:

```
SOM_Scope dirEntry  SOMLINK dropEntry(phoneDir somSelf, string entryID)
{
    int probe;
    char *probename;
    dirEntry dptr;
    Environment *ev;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","dropEntry");

    ev = SOM_CreateLocalEnvironment();
    for (probe = 0;probe < sequenceLength(_directory);probe++) {
        probename = dirEntry__get_name(sequenceElement(_directory,
                                       probe));
        if (strcmp(probename, entryID) == 0) {
            dptr = sequenceElement(_directory, probe);
            sequenceLength(_directory)--;
            for (; probe < sequenceLength(_directory); probe++) {
                sequenceElement(_directory, probe) =
                sequenceElement(_directory, (probe + 1));
            }
            _sompSetDirty (somSelf, ev);
            SOM_DestroyLocalEnvironment(ev);
            return dptr;
        } /* endif */
    } /* endfor */

    SOM_DestroyLocalEnvironment(ev);
    return (dirEntry) NULL;
}
```

Dirty Objects: Observe that the implementation of the persistent “phoneDir” class has the following line added to each method that modifies a persistent object:

```
_sompSetDirty (somSelf, env);
```

This line invokes the **sompSetDirty** method. The Persistence Framework can be optimized to write only “dirty” objects (objects that have been modified since they were last written) whenever possible. Therefore, any method that updates the persistent data of an object should use the **sompSetDirty** method to mark the object as dirty. Clean objects may also be stored at the discretion of the Persistence Framework.

Recall that **sompIsDirty** by default always returns TRUE. In the implementation of the **sompIsDirty** method below, **sompIsDirty** returns the result of **sompGetDirty**. **sompGetDirty** reports TRUE if **sompSetDirty** has been invoked on the object.

```
SOM_Scope boolean  SOMLINK sompIsDirty(phoneDir somSelf,
Environment *ev)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","sompIsDirty");

    return (_sompGetDirty(somSelf, ev));
}
```

Below, the additions to **somInit** and **somUninit** are shown in bold. These additions must be made to your persistent object if your object overrides **somInit** and **somUninit**. These methods perform initialization and cleanup that must be completed for the framework to work correctly.

```
SOM_Scope void  SOMLINK somInit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somInit");

    phoneDir_parent_SOMPPersistentObject_somInit(somSelf);
    sequenceMaximum(_directory) = MAXDIRSIZE;
    sequenceLength(_directory) = 0;
    _directory._buffer =
        SOMMalloc(sizeof (dirEntry) * MAXDIRSIZE);
}

SOM_Scope void  SOMLINK somUninit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somUninit");

    if (_directory._buffer) SOMFree(_directory._buffer);
    phoneDir_parent_SOMPPersistentObject_somUninit(somSelf);
}
```

Example 3: Definition of persistent “dirEntry”

Revising “dirEntry”: The following code shows the definition of the persistent “dirEntry” class, with differences from the nonpersistent version shown in bold:

```
#include <po.idl>

interface dirEntry : SOMPPersistentObject
{
   attribute string name;
   attribute string phone;

   void mkEntry(in string name, in string phone_no);
// Initialize new entry.

   void lsEntry();
// lsEntry displays the entry on a single line.

   #ifdef __SOMIDL__
   implementation
   {
      callstyle=oidl;
      // no Environment on methods

      releaseorder: mkEntry, lsEntry,
                    _get_name, _set_name,
                    _get_phone, _set_phone;

      // Class modifiers
      filestem = direntry;

      // Attribute modifiers
      name: persistent;
      phone: persistent;

      // Method modifiers
      somInit: override;
      somUninit: override;
      sompIsDirty: override;

   };
   #endif /* __SOMIDL__ */
};
```

These changes are analogous to the changes in the “phoneDir” class. The *implementation* of the persistent “dirEntry” class is the same as for the nonpersistent “dirEntry” and includes the addition of the **sompIsDirty** implementation. One might expect to initialize the persistent objects as dirty in their **somInit** methods; however, this is unnecessary because initializing an object with a persistent ID automatically sets its state to dirty. Persistent-object initialization is discussed in a later section.

The next version of the “phoneDir” class demonstrates the multiple inheritance case.

Example 4: Multiple inheritance definition of persistent “pphoneDir”

Assuming that the original non-persistent definition of the “phoneDir” class is in the “phonedir.idl” file, then the following interface definition of “pphoneDir” demonstrates the ability to define a new persistent object class that inherits both from **SOMPPersistentObject** and from an existing base class, namely “phoneDir”. Notice that, although the “directory” attribute was defined in the base “phoneDir” class, it can be marked as persistent in the derived class.

```
#include <po.idl>
#include <phonedir.idl>

interface pphoneDir : phoneDir, SOMPPersistentObject {

   #ifdef __SOMIDL__
   implementation
   {
      callstyle=oidl;
      // Attribute modifiers
      directory: persistent;
      // Method modifiers
      addEntry: override;
      dropEntry: override;
      somInit: override;
      somUninit: override;
      sompIsDirty: override;
   };
   #endif /* __SOMIDL__ */
};
```

Because most of the implementation remains in the “phoneDir” object the implementation of pphoneDir is quite small. The overridden methods “addEntry” and “dropEntry” make use of their parent implementations and then mark the object as dirty if the parent methods were successful.

```
SOM_Scope long  SOMLINK addEntry(pphoneDir somSelf, dirEntry entry)
{
    Environment *ev;
    long rc;
    /* pphoneDirData *somThis = pphoneDirGetData(somSelf); */
    pphoneDirMethodDebug("pphoneDir","addEntry");

    rc = pphoneDir_parent_phoneDir_addEntry(somSelf, entry);
    if (rc==0) {
       ev = SOM_CreateLocalEnvironment();
       _sompSetDirty (somSelf, ev);
       SOM_DestroyLocalEnvironment(ev);
    }
    return (rc);
}
SOM_Scope dirEntry  SOMLINK dropEntry(pphoneDir somSelf, string
entryID)
{
    dirEntry dptr;
    Environment *ev;
    /* pphoneDirData *somThis = pphoneDirGetData(somSelf); */
    pphoneDirMethodDebug("pphoneDir","dropEntry");

    dptr = pphoneDir_parent_phoneDir_dropEntry(somSelf, entryID);
    if (dptr) {
       ev = SOM_CreateLocalEnvironment();
       _sompSetDirty (somSelf, ev);
       SOM_DestroyLocalEnvironment(ev);
    }
    return(dptr);
}
```

It is *very* important that you override the **somInit** and **somUninit** methods in the multiply inherited “pphoneDir” class. If you do not override these methods, then only the leftmost parent’s **somInit** and **somUninit** methods are invoked for the object when it is created and freed. The following example shows the correct implementation of the **somInit** and **somUninit** methods for the “pphoneDir” class:

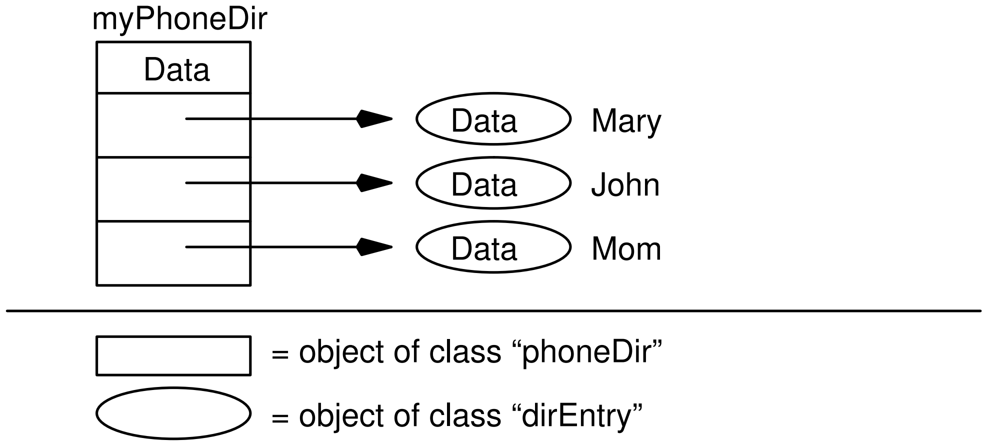
```
SOM_Scope void  SOMLINK somInit(pphoneDir somSelf)
{
    pphoneDirMethodDebug(”pphoneDir”,”somInit”);

    pphoneDir_parent_phoneDir_somInit(somSelf);
    pphoneDir_parent_SOMPPersistentObject_somInit(somSelf);
}

SOM_Scope void  SOMLINK somUninit(pphoneDir somSelf)
{
    pphoneDirMethodDebug(”pphoneDir”,”somUninit”);

    pphoneDir_parent_phoneDir_somUninit(somSelf);
    pphoneDir_parent_SOMPPersistentObject_somUninit(somSelf);
}
```

The **sompIsDirty** method override in “pphoneDir” is implemented the same as earlier examples and thus is not shown.

Embedded objects

In the phone directory example, note that the “phoneDir” class attribute “directory” is a sequence of objects. The Persistence Framework can save and restore objects that are arbitrarily complex. That is, the Persistence Framework can save and restore objects that contain pointers to other objects, which themselves contain pointers to yet other objects. For example, a “phoneDir” object might contain an sequence of pointers to “dirEntry” objects, as shown in the following illustration:

Objects pointing to other objects



The relationship between the “myPhoneDir” object and the “Mary,” “John,” and “Mom” objects is referred to here as a “parent-child” relationship. For example, “myPhoneDir” is said to be the “parent object” of “Mary,” “John,” and “Mom,” and “Mary,” “John,” and “Mom” are said to be the “children” of the “myPhoneDir” object. Parent-child relationships between *objects* are orthogonal to parent-child relationships between *classes*; "Mary" can be an instance of a class that is not related (in terms of the class derivation hierarchy) to the class of which "myPhoneDir" is an instance.

Because the "myPhoneDir" object points to the "Mary", "John", and "Mom" objects, all the objects are stored and restored at the same time, even if they are stored in different files. That is, when the programmer asks for myPhoneDir to be stored, as a consequence, "Mary", "John", and "Mom" will also be stored. Similarly, when the programmer asks for myPhoneDir to be restored, "Mary", "John", and "Mom" will be restored as well. (There are methods available whereby the programmer can request that the children of an object *not* be saved/restored along with its parent object. This will be discussed later.)

Persistent object IDs

So far, we have described only how you would go about declaring an object as persistent. In this section and the next, we describe two key characteristics of the Persistence Framework which must be understood before we show how to store your persistent objects. These are **Persistent Object ID**s and **I/O Groups**. In order to be saved and restored, each persistent object must be assigned an ID. When you assign an ID to a persistent object, this tells the Persistence Framework how and where to store your object. This also tells the Framework whether an object should be grouped with another.

A persistent ID is an object which contains a string value that uniquely identifies a persistent object. In general, a persistent ID string has the following format:

```
<IOGroupMgrClassName>:<IOGroupName>:<GroupOffset>
```

The first part of the ID string is the name of an I/O Group Manager class. This class defines how the object will be stored. The second part of the ID string is a name which is understandable to the I/O Group Manager class. The I/O Group Name defines where you want the object stored. By initializing multiple objects with the same I/O Group Manager class name and I/O Group Name, you indicate to the Persistence Framework that the objects should be grouped together into an **I/O Group**. (I/O Groups are described more in the next section.) The last part of the ID, the group offset, is used to uniquely identify an object within a group of objects that have the same I/O Group Manager class name and I/O Group name.

Consider the following specific example of a persistent ID string:

```
SOMPAscii:./pdata/phoneDir:0
```

The first part of the above persistent ID (`SOMPAscii`) identifies the class of the object's I/O Group Manager. In this case, the default I/O Group Manager class **SOMPAscii** is used. The second part (`./pdata/phoneDir`) is a name that the I/O Group Manager **SOMPAscii** understands. For the **SOMPAscii** I/O Group Manager, the second part of the ID is always a file name indicating the file in which the object will be stored. The final portion of the ID (0) is a key number (also known as the group offset number) that uniquely identifies the object within an I/O Group and therefore within the file where the persistent object will be stored.

Note: Throughout this chapter, persistent IDs are shown for AIX. For OS/2 or Windows, a persistent ID string shown as `SOMPAscii:./pdata/phoneDir:0` would be written instead as `SOMPAscii:.\\pdata\\phoneDir:0` (using double backslashes within a string in order to designate a backslash rather than the escape character, as defined in the C language).

A Persistent ID string has a maximum size of **SOMPMAXIDSIZE**. This number is defined in the main Persistence Framework header file (somp.h).

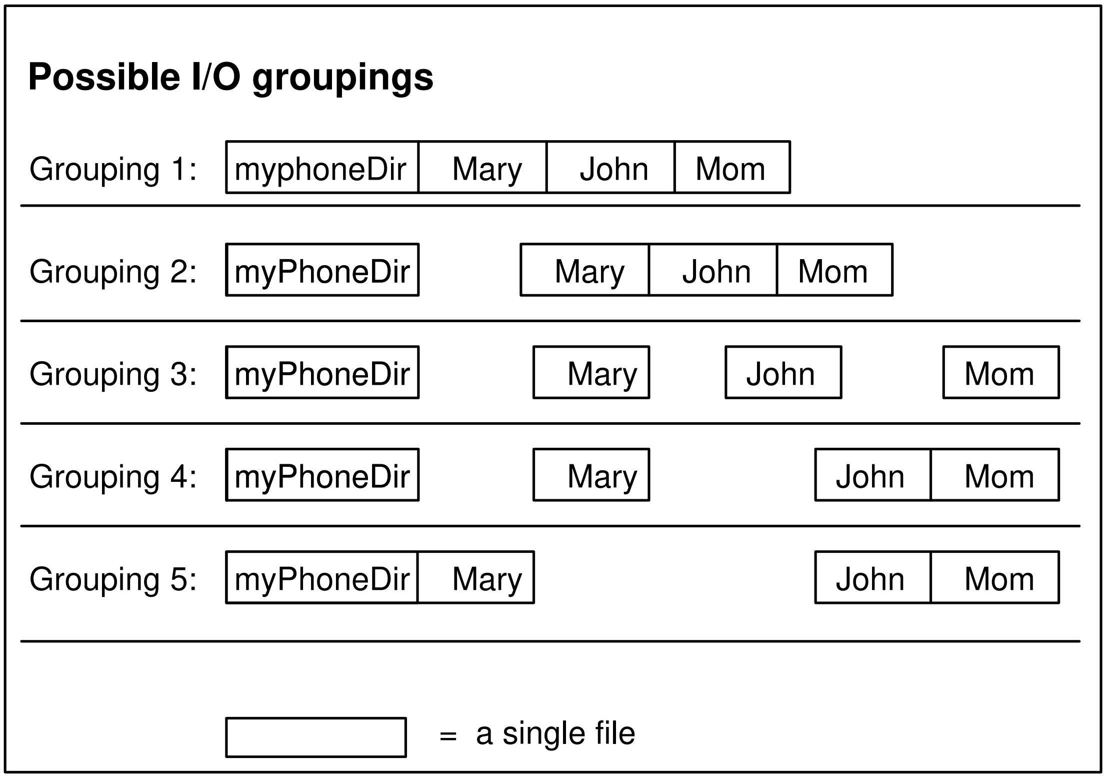
A persistent object must be associated with an ID before it can be stored/restored. Every persistent object supports methods (inherited from **SOMPPersistentObject**) for setting its persistent ID. These methods will be demonstrated later. There are several ways to create a persistent ID:

- The programmer can specifically create an ID;
- The programmer can ask an "ID Assigner" to create an ID; or
- The programmer can ask for an ID to be assigned that will place the object in the same I/O group as some other object.

To have the system assign an ID for a persistent object (the second item above), a client program must create an **ID Assigner** object. An ID Assigner is an object that knows how to manufacturer a persistent ID for an object. The Persistence Framework supplies a class, **SOMPIdAssigner**, from which ID Assigners can be instantiated.

I/O groups

All persistent objects, once initialized with a Persistent ID, are placed into an I/O Group. One I/O group consists of all the objects that are stored together in a single file. An I/O group can consist of a single object. The organization of objects into I/O groups is unrelated to their pointer relationships. Thus, the previously discussed "phoneDir" and "dirEntry" objects might have any of the following groupings, among others:



Regardless of how persistent objects are grouped, the parent-child relationships among them still hold. That is, even if the "myPhoneDir" object belongs to a different I/O group from the "Mary," "John," and "Mom" objects, whenever the "myPhoneDir" object is saved or restored, the other objects will also be saved/restored (unless the programmer specifies otherwise), because they are children of "myPhoneDir."

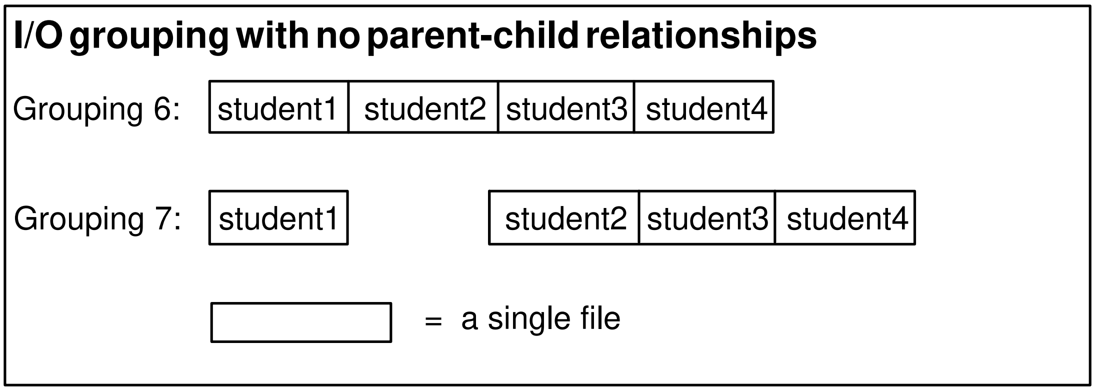
An I/O Group is an object of class **SOMPIOGroup**. You can obtain a persistent object's I/O Group by invoking the method **sompGetIOGroup** on the persistent object.

I/O Group Managers

Each I/O group is associated with exactly one I/O Group Manager object — an object whose class is derived from **SOMPIOGroupMgrAbstract**. The I/O Group Manager is determined based on the Persistent ID(s) of the object(s) contained in the I/O Group. The default I/O Group Manager class, supplied by the Persistence Framework, is **SOMPAscii**.

The **SOMPAscii** I/O Group Manager stores each I/O group as a single file. In addition to organizing persistent objects into groups, a **SOMPAscii** I/O Group Manager also maintains structure and status (“bookkeeping”) information pertaining to the persistent object(s) in the group. This bookkeeping data is stored in the I/O group as readable text. Other I/O group managers may store these data in different formats. A second I/O Group Manager class supplied by the Persistence Framework is **SOMPBinary**. The **SOMPBinary** I/O Group Manager is similar to **SOMPAscii** except that numeric data is stored in binary format .

When you store an object with either of the supplied I/O Group Managers, this also stores all of the objects in the same I/O Group if they are dirty. Consider the following groups of objects:



In the Grouping 6 of student objects, there are no parent-child relationships as in the previous example groupings; however, when “student1” is stored, the other objects (“student2”, “student3” and “student4”) are also stored. In Grouping 7, by contrast, when “student1” is stored, the other student objects will not be stored.

When you restore an object with either of the supplied I/O Group Managers, all objects in the group are instantiated and initialized, but only the data of the requested object is read. Thus, when “student1” of Grouping 6 is restored, the other students grouped with “student1” are instantiated and initialized, but their persistent data is not read. In Grouping 7, when “student1” is restored, none of the other student objects is restored.

It is possible to define your own I/O Group Manager classes by subclassing from the **SOMPIOGroupMgrAbstract** class. For complete information, refer to Appendix D, “Subclassing the Persistence Framework.”

8.4 Saving and Restoring Persistent Objects

This section describes how to save and restore persistent objects. First, we give an overview of the general approach, then we examine an example program that follows this approach to create, store, and restore a persistent telephone-directory object.

Saving a persistent object

There are five steps for creating and storing a persistent object:

1. Initialize the Persistence Framework.

   The Persistence Framework is initialized by creating a **Persistent Storage Manager** object. The Persistent Storage Manager coordinates the saving and restoring of persistent objects. It provides the primary interface for clients of the Persistence Framework. A Persistent Storage Manager is created by instantiating the **SOMPPersistentStorageMgr** class, using the SOM-defined procedure **SOMPPersistentStorageMgrNew**.

   There is only one instance of a Persistent Storage Manager object for a running process, regardless of how often you call the **SOMPPersistentStorageMgrNew** procedure. Therefore, while it may appear to be a memory leak to call **SOMPPersistentStorageMgrNew** often, it is not. You must not free (via **somFree**) the object returned from **SOMPPersistentStorageMgrNew**. If you do, the Framework will forget about previously initialized and restored objects.

   If the client program is to use system-assigned IDs for persistent objects, then initialization also includes creating a new instance of the **SOMPIdAssigner** class. The ID assigner object supports methods and data elements needed for manufacturing persistent IDs. If you instantiate a **SOMPIdAssigner** object you must also free it with **somFree**.

2. Instantiate the persistent object. (This is done in the same way that nonpersistent objects are instantiated.)

3. Assign a persistent ID to the persistent object.

   A persistent object must have a persistent ID to identify its I/O group. A later section discusses the options available for assigning IDs to objects.

4. Store the object, using the **sompStoreObject** method.

5. Remember the ID of the persistent object. To restore the object you just stored, you will need to remember the string representation of its ID. This is particularly important if the ID has been generated for you with a **SOMPIdAssigner** object. The ID string can be obtained by using the **sompGetPersistentIdString** method of the persistent object.

Restoring a persistent object

There are four steps involved in restoring a persistent object:

1. Initialize the Persistence Framework, by instantiating a Persistent Storage Manager object (as described above).

2. Set the ID of the object to be restored.

   The Persistence Framework must be supplied with the persistent ID of the persistent object to be restored. To assign an ID string to a persistent ID object:

   A. Instantiate a new persistent ID object using **SOMPPersistentIdNew**, and
   B. Assign the correct ID string to the ID object using the **somutSetIdString** method. The string value specified on **somutSetIdString** must be the same as that returned by **sompGetPersistentIdString** in step 5 above.

3. Restore the object, using the **sompRestoreObject** method.

   The **sompRestoreObject** method restores the object having the specified persistent ID, as well all of the object's persistent children.

4. Free the ID object, using the **somFree** method.

Example 5: Storing and restoring a persistent "phoneDir"

The following example program creates, saves, and restores a persistent telephone directory containing two entries. The main program invokes two functions: "save", which creates and stores the persistent phone directory, and "restore", which restores the directory object. These use the persistent telephone directory as implemented in Examples 2 and 3.

```
#include <somp.h>
#include "phonedir.h"
#include "direntry.h"

void checkError(Environment *ev);
void save(Environment *ev);
void restore(Environment *ev);

main()
{
   Environment *ev;
   ev = SOM_CreateLocalEnvironment();

   save(ev);
   restore(ev);

   SOM_DestroyLocalEnvironment(ev);
}
```

The following function is used throughout the sample to check for errors returned by the Persistence Framework.

```
void checkError(Environment *ev)
{
  sompException *params;

  if(ev->_major == NO_EXCEPTION) return;

  somPrintf("Exception %s raised.\n", somExceptionId(ev));
  if (strcmp(somExceptionId(ev), ex_SOMPError_sompException)==0) {
     params = (sompException*)somExceptionValue(ev);
     somPrintf("  SOMP primary error = %d\n", params->primary);
     somPrintf("SOMP secondary error = %d\n", params->secondary);
  }
  somExceptionFree(ev);
  exit(1);
}
```

The save function

The “save” function creates and stores the persistent phone directory:

```
void save(Environment *ev)
{
   SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
   dirEntry  name1, name2;
   phoneDir  mylist;
   SOMPPersistentId pid;

/* Create persistent phone directory.
   ---------------------------------*/
   mylist = phoneDirNew();
   pid = SOMPPersistentIdNew();
   _somutSetIdString(pid, ev, ”SOMPAscii:./pdata/phoneDir:0”);
   checkError(ev);
   _sompInitGivenId (mylist, ev, pid); /* copies the given ID */
   checkError(ev);
   _somFree (pid); /* Free the ID, mylist retains a copy */

/* Add entry 1.
   ------------*/
   name1 = dirEntryNew();
   _mkEntry  (name1, ”Roger”, ”555–5085”);
   /* put name1 in same group as mylist */
   _sompInitNearObject(name1, ev, mylist);
   checkError(ev);
   _addEntry (mylist, name1);

/* Add entry 2.
   ------------*/
   name2 = dirEntryNew();
   _mkEntry  (name2, ”Hari”, ”555–5079”);
   /* put name2 in same group as mylist */
   _sompInitNearObject (name2, ev, mylist);
   checkError(ev);
   _addEntry (mylist, name2);

/* Display phone directory.
   ------------------------*/
   _printDirInfo(mylist);

/* Store the phone directory.
   Since name1 and name2 are children of mylist, they are
   stored along with mylist.
   ------------------------------------------------------*/
   _sompStoreObject (psm, ev, mylist);
   checkError(ev);

   _somFree (name2);
   _somFree (name1);
   _somFree (mylist);
}
```

The code that assigns the “phoneDir” object (“mylist”) its persistent ID is shown in bold. The persistent ID consists of three parts, delimited by colons and concatenated into a single string, as follows:

- For this example, the I/O Group Manager class is **SOMPAscii**, the default I/O group manager supplied with the Persistence Framework for storing persistent objects in files.
- The pathname for the object’s group is “./pdata/phoneDir”, which indicates that the I/O group “phoneDir” is stored in the “pdata” subdirectory of the directory in which the process is run. (This example is for AIX. For OS/2 and Windows users, the comparable pathname would be specified as “.\\pdata\\phoneDir”, using double backslashes within the string to designate a backslash rather than the escape character, as defined in the C language.)
- The offset of the object is 0. The offset number is used to uniquely identify the phone directory object within the group. The offset should not be confused as being the byte offset of the object data within the file in which the object is stored.

The persistent ID value is assigned to a persistent ID object (“pid”) using the **somutSetIdString** method:

```
_somutSetIdString(pid, ev, "SOMPAscii:./pdata/phoneDir:0");
```

and is assigned to the “mylist” persistent object using the **sompInitGivenId** method:

```
_sompInitGivenId (mylist, ev, pid);
```

Persistent IDs are assigned to the two directory-entry objects using the **sompInitNearObject** method:

```
_sompInitNearObject(name1, ev, mylist);
```

This method assigns a persistent ID to the “name1” object, which is equivalent to the “mylist” object ID except for the persistent ID offset number. The **sompInitNearObject** method will assign a new offset number for the “name1” ID. You can determine the ID string of the directory entry objects with the **sompGetPersistentIdString** method. By initializing the directory entry objects with **sompInitNearObject**, they become grouped with the phone directory object. For the **SOMPAscii** I/O Group Manager class that will be used to store these objects, that means that all the objects will be stored in the same file.

The phone directory and both of the directory entries are stored by the method **sompStoreObject**. Since the two “dirEntry” objects are children of (pointed to by) the “phoneDir” object, all three are stored by a single invocation of the **sompStoreObject** method.

Finally, the persistent ID object (“pid”) is freed, using the **somFree** method. The persistent ID you pass to the **sompInitGivenId** is copied; therefore, to avoid a memory leak, you must free the ID you allocated.

The restore function

The “restore” function restores the persistent phone directory:

```
void restore(Environment *ev)
{
    SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
    SOMPPersistentId pid;
    phoneDir  mylist;

    InitPhoneLib();

 /* Set persistent ID.
    ------------------ */
    pid  = SOMPPersistentIdNew();
    _somutSetIdString(pid, ev, "SOMPAscii:./pdata/phoneDir:0");
    checkError(ev);

/*  Restore the Directory.
    ---------------------- */
    mylist = _sompRestoreObject (psm, ev, pid);
    checkError(ev);

/*  Display the directory.
    ----------------------*/
    _printDirInfo(mylist);

    _somFree (pid);
}
```

Notice the call to the “InitPhoneLib” function. This serves as a reminder that, before you can restore objects with the Persistence Framework, the class objects of the objects you will be restoring must exist. The implementation of “InitPhoneLib” is specific to how the implementation of the “phoneDir” and “dirEntry” classes is linked with the client program. If the classes are *statically* linked with the “restore” function, “InitPhoneLib” should instantiate each *class object* involved in the restore (_phoneDir and _dirEntry). In this case, “InitPhoneLib” could be written as

```
void InitPhoneLib() {
  phoneDirNewClass(0,0);
  dirEntryNewClass(0,0);
}
```

If the client program is *not* statically linked with the implementation of the “phoneDir” and “dirEntry” classes, the call to “InitPhoneLib” can be omitted. The Persistence Framework will dynamically load the classes and create their class objects (via **somFindClass**), assuming that the *dllname* modifier of the classes correctly specifies the name of the DLL containing the class implementation or the default DLL name (the class name) is applicable. For more information on building dynamically loadable class libraries, see “Creating a SOM Class Library” in Chapter 5, “Implementing Classes in SOM.”

The Persistence Framework must be supplied with the persistent ID of the persistent object to be restored. The ID value includes both the complete pathname used for storing the phone directory and the offset number (0) of the phone directory object. To assign the ID string to a persistent object ID, two steps are required:

A. Instantiate a new persistent ID (“pid”) with **SOMPPersistentIdNew**, and
B. Assign the correct ID value to “pid” with the **somutSetIdString** method.

The phone directory is restored using the **sompRestoreObject** method. As the name implies, this method restores the object specified by “pid”, as well all of the object’s persistent children. In this example, the first object of the I/O group (offset 0) was specified by "pid." Since the object stored at offset 0 is an instance of the "phoneDir" class, and the instances of the "dirEntry" class are its children, the complete phone directory is restored by a single invocation of the **sompRestoreObject** method.

After the directory object is restored, its contents are displayed using the method "printDirInfo". The persistent ID is then freed (using **somFree**) prior to termination of the "restore()" function.

Unlike the "save" function, the "restore" function does not explicitly instantiate "phoneDir" and "dirEntry" objects. Instead, the necessary instances are created by the Persistence Framework when the phone directory is restored.

Persistent Object ID initialization

There are three ways a persistent object can have its ID initialized. An object can be explicitly given an ID, an object can be given a system-assigned ID, or an object can be assigned an ID near some other specified object. The three different methods for Persistent Object ID initialization are **sompInitGivenId, sompInitNextAvail,** and **sompInitNearObject,** respectively.

Initialization with given ID

To initialize a persistent object with a given ID, follow these steps:

- Instantiate a **SOMPPersistentId** object,
- Tell the **SOMPPersistentId** object what its ID string is, and
- Tell the Persistent Object to use this ID, using the **sompInitGivenId** method.

The following example code demonstrates this process:

```
 PersistentDog dog = PersistentDogNew();
 SOMPPersistentId pid = SOMPPersistentIdNew();
 _somutSetIdString
      (pid, ev, "SOMPAscii:/u/roger/dogs/dog1.dog:0");
_sompInitGivenId(dog, env, pid);
```

There are a number of methods for telling an ID what its ID string is:

somutSetIdString To give the complete ID string (that specifies the I/O Group Manager class name, the group name and the object offset number), and

sompSetIOGroupName To give only the group name. The I/O Group Manager class name and offset number remain unchanged and, if never previously set, default to **SOMPAscii** and 0 respectively.

The following example code demonstrates the use of **sompSetIOGroupName**:

```
 PersistentDog dog = PersistentDogNew();
 SOMPPersistentId pid = SOMPPersistentIdNew();
_sompSetIOGroupName(pid, ev, "/u/roger/dogs/dog1.dog");
_sompInitGivenId(dog, ev, pid);
```

Notice that these two code examples differ only in the string passed to **somutSetIdString** or **sompSetIOGroupName**.

You can specify the other two parts of the persistent ID string independently with **sompSetIOGroupMgrClassName** and **sompSetGroupOffset**. For example, each part of the ID string could be set independently, as follows:

```
 PersistentDog dog = PersistentDogNew();
 SOMPPersistentId pid = SOMPPersistentIdNew();
_sompSetIOGroupMgrClassName(pid, ev, "SOMPBinary");
_sompSetIOGroupName(pid, ev, "/u/roger/dogs/dog1.dog");
_sompSetGroupOffset(pid, ev, 0);
_sompInitGivenId(dog, ev, pid);
```

Environment variables in pathnames

Environment variables can be used in the Group Name part of a Persistent ID string. When they are used, they are expanded at store and restore time to determine actual values. This is primarily intended to allow programs to easily move objects from one directory to another. The following code initializes an object with a known persistent ID containing an environment variable:

```
PersistentDog dog = PersistentDogNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
_somutSetIdString
 (pid, ev, ”SOMPBinary:\”DOGDIR\”/dogs/dog1.dog:0”);
_sompInitGivenId(dog, ev, pid);
```

Note that the environment variable is quoted, and that a backslash is used to insert a quote character in a string.

The Persistence Framework will then expand DOGDIR just before storing (or restoring), and the resulting ID becomes the final ID. For example, if DOGDIR is set to /u/roger, this previous code would be equivalent to the following code:

```
PersistentDog dog = PersistentDogNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
_somutSetIdString
 (pid, ev, ”SOMPBinary:/u/roger/dogs/dog1.dog:0”);
_sompInitGivenId(dog, ev, pid);
```

The environment variable can be used any place within the group name portion of the ID string.

Initialization with next available ID

In this mode of ID initialization, the **SOMPIdAssigner** object assigns a persistent object the next available persistent ID. The steps necessary to assign a persistent ID using the **SOMPIdAssigner** object are as follows:

- Instantiate an ID Assigner object, and
- Initialize the persistent object, with method **sompInitNextAvail,** passing the instantiated ID Assigner as a parameter.

The following code demonstrates these steps:

```
PersistentDog dog = PersistentDogNew();
SOMPIdAssigner systemIdAssigner = SOMPIdAssignerNew();
_sompInitNextAvail(dog, ev, systemIdAssigner);
```

The default **SOMPIdAssigner** makes use of the environment variable SOMP_PERSIST. If SOMP_PERSIST is set to a directory path, the **SOMPIdAssigner** will create IDs so that persistent objects with those IDs will be stored in the named directory.

The algorithm used by **SOMPIdAssigner** to select the next available persistent ID is as follows:

```
If (environment variable SOMP_PERSIST is set)
     storage directory is value  of SOMP_PERSIST
else storage directory is current directory.

if (file somplast.id exists in storage directory)
    read somplast.id to find last assigned ID number
else {
    create somplast.id
    assume last assigned ID number is 0
}
id = SOMPAscii:/storage_dir/p<last assigned id in hex>:0
increment last assigned ID in file somplast.id
```

Because the names of the files produced by the supplied **SOMPIdAssigner** object are limited to eight bytes in length, the algorithm is limited to producing at most 268,435,455 file names.

To change the supplied algorithm, developers can derive a new class from **SOMPIdAssigner** and override the method **sompGetSystemAssignedId**.

Initialization near another object

Initialization near another object is perhaps the most common mode of persistent ID initialization. This mode assumes that some object knows its persistent ID, and another object will be assigned a persistent ID so that it resides in the same I/O group. The following code demonstrates this technique:

```
   PersistentDog dog1, dog2;
   SOMPIdAssigner systemIdAssigner;

/* Instantiate dogs. */
   dog1 = PersistentDogNew();
   dog2 = PersistentDogNew();

/* Instantiate an ID Assigner. */
   systemIdAssigner = SOMPIdAssignerNew();

/* Have system assign ID. */
   _sompInitNextAvail(dog1, ev, systemIdAssigner);

/* Place dog 2 in same IO Group as  dog 1. */
   _sompInitNearObject(dog2, ev, dog1);
```

The ID string assigned to the objects initialized with **sompInitNextAvail** can be determined with the **sompGetPersistentIdString** method as shown:

```
string idBuffer[SOMPMAXIDSIZE];

_sompGetPersistentIdString(dog1, ev, idBuffer);
somPrintf("dog1 ID string is : %s\n", idBuffer);
_sompGetPersistentIdString(dog2, ev, idBuffer);
somPrintf("dog2 ID string is : %s\n", idBuffer);
```

Assuming that the environment variable SOMP_PERSIST has not been set, the ID strings output by the above example would be:

```
dog1 ID string is : SOMPAscii:./p00000000:0
dog2 ID string is : SOMPAscii:./p00000000:1
```

It is also possible to initialize objects "near" to one another by initializing them with the same ID with a change made to the group offset number between initializations. In the example below, dog0 and dog1 are placed in the same I/O Group.

```
PersistentDog dog0 = PersistentDogNew();
PersistentDog dog1 = PersistentDogNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
_somutSetIdString
      (pid, ev, "SOMPAscii:./dogs/dog1.dog:0");
_sompInitGivenId(dog0, env, pid);
_sompSetGroupOffset(pid, ev, 1);/* change offset */
_sompInitGivenId(dog1, ev, pid);
```

Example 6: Storing objects in multiple files using system-assigned IDs

This example is identical to Example 5, except that the two “dirEntry” objects and the “phoneDir” objects are all stored in different files, and an ID Assigner is used to create system-assigned persistent IDs. The definitions and implementations of “phoneDir” and “dirEntry” are unchanged.

The test program is unchanged except that the “save” method differs, as shown below (with significant differences shown in bold):

```
void save(Environment *ev)
{
   SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
   SOMPIdAssigner idAssigner = SOMPIdAssignerNew();
   dirEntry  name1, name2;
   phoneDir  mylist;

/* Create phone directory. */
   mylist = phoneDirNew();
   _sompInitNextAvail (mylist, ev, idAssigner);
   checkError(ev);

/* Add entry 1. */
   name1 = dirEntryNew();
   _mkEntry  (name1, ”Roger”, ”555–5085”);
   _sompInitNextAvail (name1, ev, idAssigner);
   checkError(ev);
   _addEntry (mylist, name1);

/* Add entry 2. */
   name2 = dirEntryNew();
   _mkEntry  (name2, ”Hari”, ”555–5079”);
   _sompInitNextAvail (name2, ev, idAssigner);
   checkError(ev);
   _addEntry (mylist, name2);

/* Display phone directory. */
   _printDirInfo(mylist);

/* Store the phone directory. */
   _sompStoreObject (psm, ev, mylist);
   checkError(ev);

   _somFree (idAssigner);
   _somFree (name2);
   _somFree (name1);
   _somFree (mylist);
}
```

The main difference from the previous example is that each of the three persistent objects is stored in a separate I/O group (file). This is accomplished by using the system ID Assigner to assign new IDs through the **sompInitNextAvail** method. This can be contrasted to the previous example in which only one group (for “mylist”) was used, and the IDs for “name1” and “name2” were initialized as “near objects” of “mylist” (that is, objects to be stored in the same I/O group with “mylist”). This difference only changes the physical storage arrangement of the phone directory and its entries. The “name1” and “name2” objects are still children of “mylist”; thus, all three objects are still stored by a single invocation of **sompStoreObject**.

The previous examples have demonstrated storing multiple persistent objects in a single I/O group, and storing persistent objects in multiple I/O groups, one object per group. These techniques can be combined to produce other arrangements, such as multiple groups, each containing multiple objects. For example, assume that six persistent objects, “p0” through “p5”, are to be stored in two groups. The first group is to contain objects p0 through p2, and the second group objects p3 through p5. This would be accomplished by initializing the persistent IDs for the objects in the following way:

```
/* For the first I/O group: */
   SOMPPersistentId pid;
   _sompGetSystemAssignedId(idAssigner, ev, &pid);
   _sompInitGivenId (p0, ev, pid);
   _sompInitNearObject (p1, ev, p0);
   _sompInitNearObject (p2, ev, p0);
   _somFree(pid);

/* For the second I/O group */
   _sompGetSystemAssignedId(idAssigner, ev, &pid);
   _sompInitGivenId (p3, ev, pid);
   _sompInitNearObject (p4, ev, p3);
   _sompInitNearObject (p5, ev, p3);
   _somFree(pid);
```

Notice the highlighted line above which creates a new system assigned persistent ID. In the previous example, the invocation of **sompGetSystemAssignedId** was made for you by **sompInitNextAvail**. When you invoke **sompGetSystemAssignedId** directly, you must pass the address of an ID object. When the method completes successfully, the ID (pid) contains a newly instantiated persistent ID object. Notice also that the caller is responsible for freeing the returned ID.

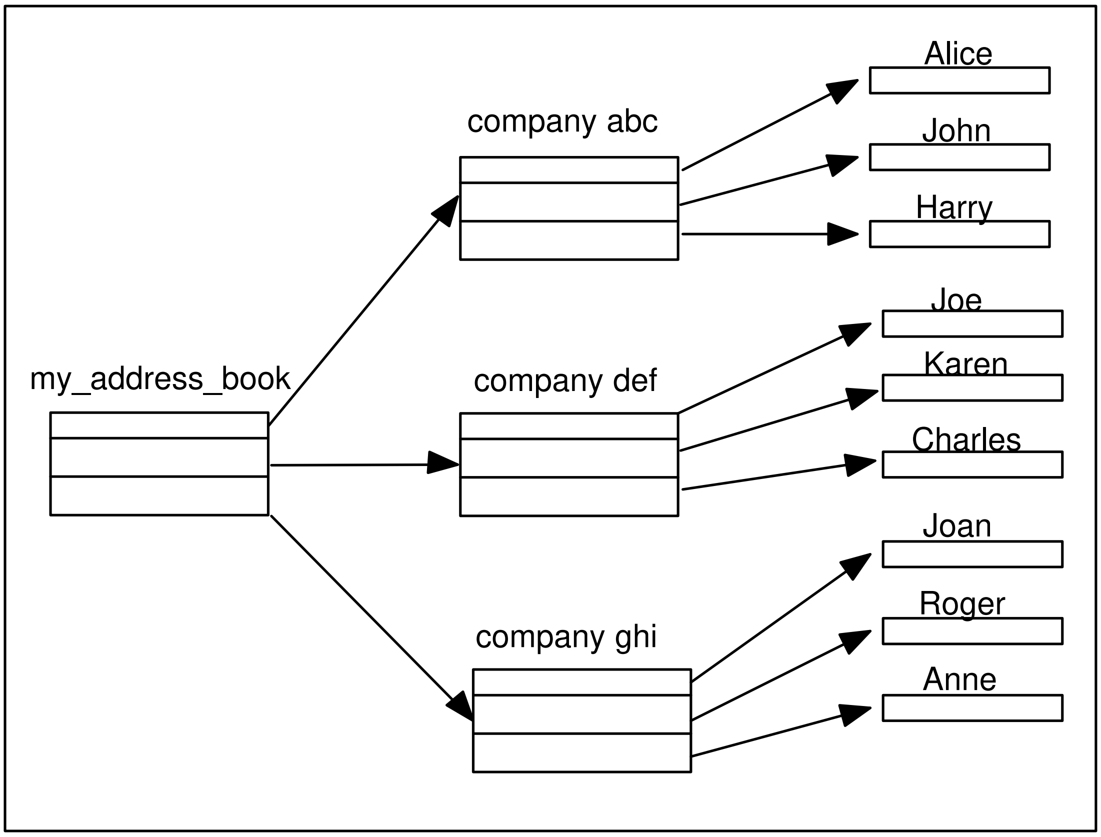
By extension of these techniques, an almost limitless variety of storage groupings can be arranged.

Another difference between this example and the previous example is the use of system-assigned persistent IDs, rather than application-assigned IDs. While the use of system-assigned IDs is often more convenient, the use of application-assigned IDs provides greater application control over object storage. By placing I/O group pathnames under application control, persistent objects can be stored in any file system directory without the need to alter the SOMP_PERSIST environment variable prior to process execution.

Application-assigned IDs have another advantage over system-assigned IDs, which can be seen by comparing the results of multiple runs of the two example programs. When the previous example (using system-assigned IDs) is run repeatedly, a new copy of the persistent object is created on each run. When the application-assigned ID is employed (as in the first example), each run of the program overwrites the previous version of the persistent object. When replacement of an existing persistent object is desired, application-assigned IDs are appropriate, rather than system-assigned IDs.

Read/Write without children

There are two modes to reading and writing. The first is reading/writing a whole object hierarchy. This is the standard read/write mode. The second is reading/writing an object without following persistent pointers. The second is called “read/write without children” mode. The first mode has been demonstrated in the previous two examples. Read/write without children is useful with a large hierarchy of objects, when only a part of the hierarchy needs to be read. For example, consider the following object hierarchy:



Assume here that “my_address_book” is an object of class “addressBook” that supports the method “findCompany,” which takes a string and returns a pointer to a “company” object. Assume that “company” objects support the method “findPerson,” which takes a string and returns a pointer to a “person” object. The standard mode read would be as follows:

```
SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
phoneDir  mylist;

_somutSetIdString(pid, ev,
    ”SOMPAscii:/u/roger/data/myPhoneBook:0”);
checkError(ev);
mylist = _sompRestoreObject (psm, ev, pid);
checkError(ev);
```

At the end of the invocation of **sompRestoreObject,** the entire hierarchy of objects will have been restored. It is then valid to ask the address book for the company named “def”, and then to ask that object for the person “Karen” without any further calls to the Persistence Framework.

Suppose, however, that “my_address_book” has pointers to a dozen companies, and each company has pointers to hundreds of people. Suppose further that we want to find “Roger” in company “ghi”. Because we really don’t want to read in the entire hierarchy in this case, we can specify how to traverse the hierarchy.

To avoid traversing the hierarchy, use the **sompRestoreObjectWithoutChildren** method. When reading an object without children the Persistence Framework will first read in the requested object, and then for each persistent child object referenced by the restore object it will:

- Instantiate an object of the correct type,
- Initialize that object with its correct ID, and
- Set the persistent pointer in the parent object.

The Framework will *not* read in the data for the child objects. The result of this is that the initial object has all its persistent object pointers set to objects which have the correct IDs and are the correct type, but which have not had their data read. An object which is the correct type and knows its ID but which has not been read from storage is called an *unstable* object (its state is SOMP_STATE_UNSTABLE). Before attempting to use an unstable object, you must ask the Persistent Storage Manager to restore the object, either with or without children.

Code that follows the pointers from “my_address_book” to company “def” to person “Karen” would look like this example:

```
SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
phoneDir mylist;
company myCompany;
person myPerson;

_somutSetIdString(pid, ev,
                ”SOMPAscii:/u/roger/data/myPhoneBook:0”);
checkError(ev);
mylist = _sompRestoreObjectWithoutChildren (psm, ev, pid);
checkError(ev);

myCompany = _findCompany(myList, ev, ”def”);
_sompRestoreObjectWithoutChildren(psm, ev,
    _sompGetPersistentId(myCompany, ev));
checkError(ev);

myPerson = _findPerson(myCompany, ev, ”Karen”);
_sompRestoreObjectWithoutChildren(psm, ev,
    _sompGetPersistentId(myPerson, ev));
checkError(ev);
```

In the line

```
_sompRestoreObjectWithoutChildren(psm, ev,
    _sompGetPersistentId(myCompany, ev));
```

it is not necessary to check the returned pointer (myCompany), since it will be unchanged from before the call. Before the call, the pointer points to an unstable object, whereas after the call, it points to a stable object.

The Persistence Framework also supports *writing* an object without its children. When storing an object without children, the object itself is stored, but none of the persistent pointers are followed. This is useful when a hierarchy contains many objects, but only a few are to be stored. This is accomplished with **sompStoreObjectWithoutChildren**.

Note: When child objects are grouped with their parent object (by initializing them near the parent) so that they are stored in the same file, **sompStoreObjectWithoutChildren** *will* store the children. This is because the **SOMPAscii** and **SOMPBinary** I/O Group Manager classes store a group at a time. You must initialize children into a different I/O Group if you do not want them stored.

Choosing I/O Group Manager SOMPAscii or SOMPBinary

The Persistence Framework provides two **I/O Group Managers**. The ASCII version (**SOMPAscii**, defined in fsagm.idl) stores all data in ASCII format. This allows the resulting data file to be viewed in a text editor. Although you can view the data, you should not modify the data with a text editor. Because **SOMPAscii** stores persistent object data as one continuous line, larger files may not be readable by some text editors.

Also included is a Binary version (**SOMPBinary**, defined in fsgm.idl), which stores data in binary format. These files cannot be viewed in a text editor, but are more efficient for I/O. It is advisable to use the **SOMPAscii** I/O Group Manager during initial application development and the **SOMPBinary** I/O Group Manager when preparing the final code.

Programmers choose which **IOGroupManager** to use by giving the appropriate name in the Persistent ID string for an object. As an example, the following code sets up a Persistent Object with the **SOMPAscii** I/OGroup Manager**:**

```
    SOMPPersistentId pid = SOMPPersistentIdNew();
    phoneDir mylist = phoneDirNew();
    char idBuff[SOMPMAXIDSIZE];

 /* Use Ascii IOGroupMgr:
    --------------------- */
    strcpy(idBuff, "SOMPAscii:./pdata/phoneDir:0");
    _somutSetIdString(pid, ev, idBuff);

    _sompInitGivenId (mylist, ev, pid);
    checkError(ev);
    _somFree(pid);
    _sompStoreObject (psm, ev, mylist);
```

The following segment uses the Binary **IOGroupMgr**. Note the only change is to the line in bold.

```
    SOMPPersistentId pid = SOMPPersistentIdNew();
    phoneDir  mylist = phoneDirNew();
    char idBuff[SOMPMAXIDSIZE];

 /* Use Binary IOGroupMgr:
    ---------------------- */
    strcpy(idBuff, "SOMPBinary:./pdata/phoneDir:0");
    _somutSetIdString(pid, ev, idBuff);

    _sompInitGivenId (mylist, ev, pid);
    checkError(ev);
    _somFree(pid);
    _sompStoreObject (psm, ev, mylist);
```

SOMPAscii and SOMPBinary characteristics

The supplied I/O Group Manager classes **SOMPAscii** and **SOMPBinary** exhibit the following characteristics. A user-written I/O Group Manager may or may not emulate these characteristics.

Store characteristics

When you store an object with either of the supplied **SOMPAscii** or **SOMPBinary** I/O Group Managers, this also stores all of the objects in the same I/O Group as the stored object, if the **sompIsDirty** method on those objects returns TRUE. Consider the following example which stores several objects grouped together. In the example, three “dirEntry” objects are created and initialized near to one another. While there are no parent-child object relationships between these objects, the single invocation of **sompStoreObject** will store all three objects into the file “entries”.

Invoking **sompStoreObject** for the other two objects (“name2” and “name3”) is unnecessary; in fact, if you haven’t overridden **sompIsDirty** in your object implementation, this would cause all of the objects to be stored again a second and third time. The Framework clears the “dirty” flag after an object has been stored but, by default, **sompIsDirty** returns TRUE whether the dirty flag is set or not.

```
   SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
   SOMPPersistentId pid = SOMPPersistentIdNew();
   dirEntry  name1, name2, name3;
   char idBuff[SOMPMAXIDSIZE];

   name1 = dirEntryNew();
   _mkEntry  (name1, ”Roger”, ”555-5085”);
   strcpy(idBuff, ”SOMPAscii:entries:0”);
   _somutSetIdString(pid, ev, idBuff);
   _sompInitGivenId (name1, ev, pid);
   checkError(ev);

   name2 = dirEntryNew();
   _mkEntry  (name2, ”Hari”, ”555-5079”);
   _sompSetGroupOffset(pid, ev, 10);
   _sompInitGivenId (name2, ev, pid);
   checkError(ev);

   name3 = dirEntryNew();
   _mkEntry  (name3, ”Louie”, ”555-5080”);
   _sompSetGroupOffset(pid, ev, 100);
   _sompInitGivenId (name3, ev, pid);
   checkError(ev);
   _somFree(pid);

/* Store all three dirEntry objects. */
   _sompStoreObject (psm, ev, name1);
   checkError(ev);
```

Note in the example above that the objects are considered “near” to one another and therefore in the same I/O Group, even though they were initialized with **sompInitGivenId**. This happens because the only difference between the objects’s Persistent IDs is the offset number that was set with **sompSetGroupOffset**.

Restore characteristics

When a persistent object is restored using the supplied **SOMPAscii** or **SOMPBinary** I/O Group Managers, the other objects grouped with the requested object are created and initialized with a Persistent ID by the Framework; however, only the requested object’s persistent data is read. The other objects remain in an *unstable* state. Refer to the topic “Persistent object states” below for more information on the states of a persistent object.

The following example shows how to restore the objects in the “entries” file built by the previous example.

```
    SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
    SOMPPersistentId pid = SOMPPersistentIdNew();
    dirEntry  name1, name2, name3;
    char idBuff[SOMPMAXIDSIZE];

 /* Ensure class object of restored objects exists */
    dirEntryNewClass(0, 0);

    strcpy(idBuff, ”SOMPAscii:entries:0”);
    _somutSetIdString(pid, ev, idBuff);
    checkError(ev);

 /* Restore the first object in the group. */
    name1 = _sompRestoreObject(psm, ev, pid);
    checkError(ev);
    _lsEntry(name1); /* Print out restored object */
    _somDumpSelf(_sompGetIOGroup(name1, ev), 0);

 /* Restore the second object in the group. */
    _sompSetGroupOffset(pid, ev, 10);
    name2 = _sompRestoreObject(psm, ev, pid);
    checkError(ev);
    _lsEntry(name2); /* Print out restored object */
    _somDumpSelf(_sompGetIOGroup(name2, ev), 0);

 /* Restore the third object in the group. */
    _sompSetGroupOffset(pid, ev, 100);
    name3 = _sompRestoreObject(psm, ev, pid);
    checkError(ev);
    _lsEntry(name3); /* Print out restored object */
    _somDumpSelf(_sompGetIOGroup(name3, ev), 0);
```

The example dumps the restored I/O Group with **somDumpSelf** after each restore. In the output produced by the example below, notice that all of the objects exist after the first restore. However, only the object requested for the restore has its data read and is then considered to be in a stable state.

```
Roger               555-5085
{An instance of class SOMPIOGroup at address 200A3B40
SOMPAscii:entries:0
SOMP_STATE_STABLE
SOMPAscii:entries:10
SOMP_STATE_UNSTABLE
SOMPAscii:entries:100
SOMP_STATE_UNSTABLE
}
Hari                555-5079
{An instance of class SOMPIOGroup at address 200A3B40
SOMPAscii:entries:0
SOMP_STATE_STABLE
SOMPAscii:entries:10
SOMP_STATE_STABLE
SOMPAscii:entries:100
SOMP_STATE_UNSTABLE
}
Louie               555-5080
{An instance of class SOMPIOGroup at address 200A3B40
SOMPAscii:entries:0
SOMP_STATE_STABLE
SOMPAscii:entries:10
SOMP_STATE_STABLE
SOMPAscii:entries:100
SOMP_STATE_STABLE
}
```

Modifying an object previously stored with SOMPAscii or SOMPBinary

To modify an object previously stored with either **SOMPAscii** or **SOMPBinary**, you must first restore the object. Consider the previous example. Suppose you want to change Roger's phone number to 555–5086. The following code would accomplish this:

```
   Environment *ev;
   SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
   SOMPPersistentId pid = SOMPPersistentIdNew();
   dirEntry  name1, name2, name3;
   char idBuff[SOMPMAXIDSIZE];

   ev = SOM_CreateLocalEnvironment();

/* Ensure class object of restored object exists */
   dirEntryNewClass(0, 0);

   strcpy(idBuff, "SOMPAscii:entries:0");
   _somutSetIdString(pid, ev, idBuff);
   checkError(ev);

/* Restore the first object in the group. */
   name1 = _sompRestoreObject(psm, ev, pid);
   checkError(ev);

/* Modify object */
   _mkEntry  (name1, "Roger", "555-5086");
   _sompSetDirty(name1, ev);
   _lsEntry(name1); /* Print out restored object */
   _sompStoreObject (psm, ev, name1);
   checkError(ev);

   SOM_DestroyLocalEnvironment(ev);
```

You must mark the modified object as “dirty” with **sompSetDirty** for the object to be stored if **sompIsDirty** has been overridden. Typically, this would be done for you by the persistent object implementation and in this example, by the “mkEntry” method. Here, we explicitly do it to make it obvious.

If, instead of first restoring the “Roger” object, you were to instantiate a new object with the same Persistent ID string as the stored “Roger” object, set its phone number to the new number 555–5086, and then store the object, this would effectively delete any other objects stored with “Roger”. In the example, this means that the “Hari” and “Louie” objects would be destroyed.

*Adding an object to an existing group stored with **SOMPAscii** or **SOMPBinary***

Adding an object to an existing file is similar to modifying an object. To add an object, you must first restore at least one of the objects that is already in the file. The following example adds a new “dirEntry” object “Chuck”, with offset number 1000, to the other “dirEntry” objects in the “entries” file.

```
/* Ensure class object of restored object exists */
   dirEntryNewClass(0, 0);

   strcpy(idBuff, ”SOMPAscii:entries:0”);
   _somutSetIdString(pid, ev, idBuff);
   checkError(ev);

/* Restore the first object in the group. */
   name1 = _sompRestoreObject(psm, ev, pid);
   checkError(ev);
   _lsEntry(name1); /* Print restored object */

   name2 = dirEntryNew();
   _mkEntry  (name2, ”Chuck”, ”555–5017”);
   _sompSetGroupOffset(pid, ev, 1000);
   _sompInitGivenId (name2, ev, pid);
   checkError(ev);

   _sompStoreObject (psm, ev, name2);
   checkError(ev);
```

*Files created by **SOMPAscii** and **SOMPBinary***

The **SOMPAscii** I/O Group Manager reads and writes to the file system using the **SOMPAsciiMediaInterface** class. Similarly, the **SOMPBinary** class uses the **SOMPBinaryFileMedia** class. Both classes open the files they build for exclusive read/write access (by initializing their Media Interface with **sompInitReadWrite**). If multiple processes using the Persistence Framework attempt to access the same file at the same time, the secondary request(s) are blocked. The Media Interface classes will not immediately report this condition as an error, since most file accesses are for a relatively short period of time. Instead, they will try again every three seconds for up to one minute.

The number of retries and the interval (in seconds) between retries can be changed via the two environment variables SOMP_RETRY and SOMP_RETRYI. If not set, SOMP_RETRY defaults to 20 and SOMP_RETRYI defaults to 3.

Activation and passivation

Frequently an object has both persistent and nonpersistent data. In some cases, the persistent data does not require constant maintenance and only must be brought up-to-date before the persistent object containing it is stored to persistent media. The **SOMPPersistentObject** class supports two methods to facilitate the movement of data between persistent and nonpersistent data. These methods are **sompActivated** and **sompPassivate**. Both methods are designed to be overridden by subclasses of **SOMPPersistentObject**.

The Persistence Framework invokes the **sompPassivate** method at store time just before writing a persistent object to persistent media. To have an object update its persistent data before being written, override the **sompPassivate** method in a subclass of **SOMPPersistentObject** to include code to set up the persistent data.

The Persistence Framework invokes the **sompActivated** method at restore time immediately after reading persistent data. To have an object update its nonpersistent data after being restored, override the **sompActivated** method in a subclass of **SOMPPersistentObject** to include code to update nonpersistent data.

These methods may also be useful if you would like to keep your persistent data in an internal, non-persistent data structure but still take advantage of the default attribute encoder/decoder class **SOMPAttrEncoderDecoder**. For example, suppose you have built a special collection class object which can contain a set of objects. Internal to the object, its collection of objects can be kept in any data structure you like; but doing so will not allow you to use the default attribute encoder/decoder. You could write an encoder/decoder (see "Storing Objects in Specialized Formats" later in this chapter), but using the default would be simpler. To use the default encoding for this example, you could define a sequence attribute "collection" and override the **sompActivated** and **sompPassivate** methods as highlighted below:

```
#include <po.idl>

interface myCollection : SOMPPersistentObject {

attribute sequence<SOMPPersistentObject> collection;

/* ... methods ... */

   #ifdef __SOMIDL__
   implementation
   {
      // Attribute modifiers
      collection: persistent;
      // Method modifiers
      somInit: override;
      somUninit: override;
      sompIsDirty: override;
      sompActivated: override;
      sompPassivate: override;
   };
   #endif /* __SOMIDL__ */
};
```

In the overridden **sompPassivate** method, you would transfer the contents of your internally defined collection to the collection sequence. The default encoder/decoder would then store the sequence of persistent objects. Likewise, after the collection sequence has been restored, your overridden **sompActivated** method would transfer the contents of the restored collection sequence to your internally defined collection.

8.5 Managing Persistent Objects

Checking persistent object existence

The Persistent Storage Manager can determine whether a persistent object exists in storage, given its persistent ID, via the **SOMPPersistentStorageMgr** method **sompObjectExists**. The following code shows how this method is used to detect whether the 0th object in file /mydogs/dog.dat exists:

```
SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
SOMPPersistentId pid = SOMPPersistentIdNew();

_somutSetIdString(pid, ev, ”SOMPAscii:/mydogs/dog.dat:0”);
if (_sompObjectExists(psm, ev, pid)) {
   somPrintf(”Oth object exists\n”);
} else {
   somPrintf(”Oth object does not exist\n”);
}
 _somFree(pid);
```

```
boolean sompObjectExists
     (in SOMPPersistentId objectID) raises(sompException);
// Checks to see if object exists.  If so, returns TRUE; otherwise,
// returns FALSE.
```

Deleting persistent objects

To delete a persistent object, use the **SOMPPersistentStorageMgr** method **sompDeleteObject**. This method removes the persistent object with the specified persistent ID from its I/O group. The following code shows how this method is used to delete the 0th object in the /mydogs/dog.dat file:

```
SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
SOMPPersistentId pid = SOMPPersistentIdNew();

_somutSetIdString(pid, ev, ”SOMPAscii:/mydogs/dog.dat:0”);
_sompDeleteObject(psm, ev, pid);
_somFree(pid);
```

Persistent object states

As persistent objects are created, initialized, stored and restored, they pass through a number of states. These states can be checked with the **sompCheckState** method. The method returns TRUE if the object is in the given state. The states are defined in file po.idl.

State **Description**

SOMP_STATE_UNDEFINED
When a persistent object is first instantiated, its state is *undefined*. A persistent object cannot be stored when it is in this state.

SOMP_STATE_STABLE
When a persistent object is initialized (with **sompInitGivenId**, etc.), its state moves from an *undefined* state to a *stable* state. Once the object is in this state, it can be stored.

SOMP_STATE_DIRTY

When a persistent object is initialized by you, it is assumed to be *dirty*. As discussed earlier in this chapter, a dirty object is one which has persistent data that has changed and should be stored.

When a persistent object is restored, it is initialized for you by the Framework. Because a freshly restored object is not yet dirty, the Framework turns the dirty state off after it has initialized the object. It is then the responsibility of your persistent object implementation to indicate that the object's persistent data has been changed. You can set the dirty state with **sompSetDirty**.

SOMP_STATE_UNSTABLE

When a persistent object is restored, it is first instantiated and initialized for you by the I/O Group Manager. At this point, the object exists but its persistent data has not yet been read. An object in this state is considered to be *unstable*. Once the object's data has been read, the state of the object is moved from an unstable state to a stable state.

Usually an object remains unstable for only a short period of time. There are, however, a couple of cases where you will encounter unstable objects which exist for a longer period of time. When you restore an object with **sompRestoreObjectWithoutChildren**, child objects are only instantiated and initialized and therefore remain unstable. Also, when restoring an object with either of the **SOMPAscii** or **SOMPBinary** I/O Group Managers, all other objects grouped in the same file with the restored object are instantiated and initialized, but their data is not read unless they are children of the originally requested object.

Garbage collection

When persistent objects are rewritten to files by the supplied I/O Group Managers, they are appended to the end of the file, and their original space is garbage. When an object is deleted, its space is not reclaimed. Thus, over time, an I/O group will accumulate unused space, which should occasionally be compacted. Clients can request I/O groups be compacted by marking any object in the I/O group for compaction.

A persistent object is marked for compaction with the **sompMarkForCompaction** method, which lets a persistent object know that, the next time it is stored, the entire I/O group in which it resides should be compacted. This slows down the store, since all objects in the I/O group must be moved within the file, but it also frees any unused space in the I/O group. Sample code for doing this follows:

```
SOMPPersistentStorageMgr psm = SOMPPersistentStorageMgrNew();
SOMPPersistentId pid = SOMPPersistentIdNew();
phoneDir mylist;

_somutSetIdString(pid, ev,
     "SOMPAscii:/u/roger/data/myPhoneBook.dat:0");
checkError(ev);
mylist = _sompRestoreObject (psm, ev, pid);
checkError(ev);
_somFree(pid);
/* ... */
_sompMarkForCompaction(mylist, ev);
checkError(ev);
_sompStoreObject(psm, ev, mylist);
checkError(ev);
```

To clean up unused space in files produced by the SOMPAscii or SOMPBinary I/O Group Manager classes, you may use a garbage collection program named “srgarbcl”. This program can be run from the command line as follows:

```
srgarbcl /u/roger/data/myPhoneBook.dat
```

The parameter given is a file produced by SOMPAscii which contains your objects. To clean up a file produced by SOMPBinary you would use the following syntax:

```
srgarbcl –iSOMPBinary /u/roger/data/myPhoneBook.dat
```

8.6 Storing Objects in Specialized Formats.

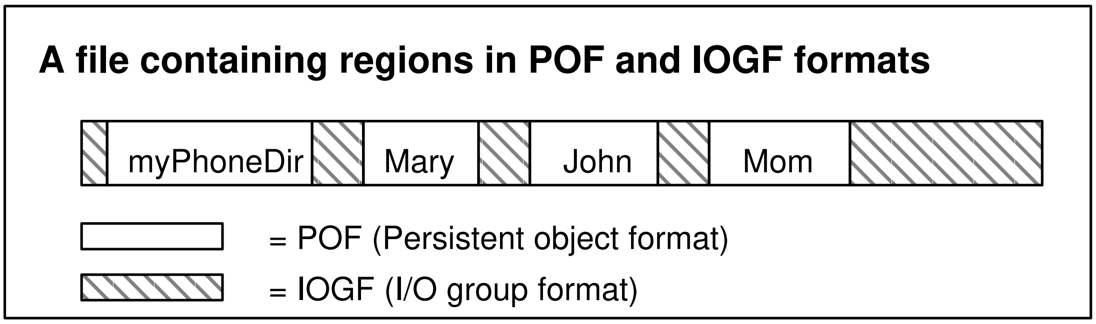
This section describes how to customize the format in which objects are written. This is done by replacing the Persistence Framework's default *Encoder/Decoder* class (**SOMPAttrEncoderDecoder**). Before discussing the role of the encoder/decoder, however, it is necessary to introduce the notion of a *persistent object format*.

Persistent object format

Persistent objects are associated with two types of formats:

- **Persistent object format (POF)** — the format of the *portion* of the file where the object's persistent data resides, and
- **I/O group format (IOGF)** — the format of the file or files which serves as a skeleton to hold together individual objects.

The Persistence Framework's supplied I/O Group Mangers build files containing regions that are formatted in IOGF (I/O group format) and regions that are formatted in POF (persistent object format), as depicted in the following illustration:



Objects that belong to the same class (such as "Mary", "John", and "Mom") need not all be written in the same POF. That is, a given class can be associated with many POFs. The choice as to which POF to use is made at run time.

Most objects can be stored in a default POF. The default POF stores each object as a series of tuples, where each tuple consists of an **attribute** name and value. The default POF requires that each element of persistent data be in the form of an **attribute** and that each persistent attribute is so noted with the **persistent** attribute modifier in the .idl file. (This .idl entry takes the form <*attributeName*> **: persistent**.) Persistent attributes may be of any valid CORBA data type.

For example, given an object with persistent attributes "name" and "phone," both of type **string**, having values "Roger" and "555–8585", the object would have the following default persistent object format:

```
2 (4)name (5)Roger (5)phone (8)555-8585
```

The respective elements of the POF are shown in bold and are described in the following text.

2 (4)name (5)Roger (5)phone (8)555-8585

The **2** indicates this record has two attributes.

2 **(4)**name (5)Roger (5)phone (8)555-8585

The **4** means the first attribute's identifier has 4 characters.

2 (4)**name** (5)Roger (5)phone (8)555-8585

The **name** indicates that the identifier of the first attribute is "name".

```
2 (4)name (5)Roger (5)phone (8)555-8585
```

The **5** means the value of the first attribute has 5 characters.

```
2 (4)name (5)Roger (5)phone (8)555-8585
```

The **Roger** indicates that the value of the "name" attribute is "Roger".

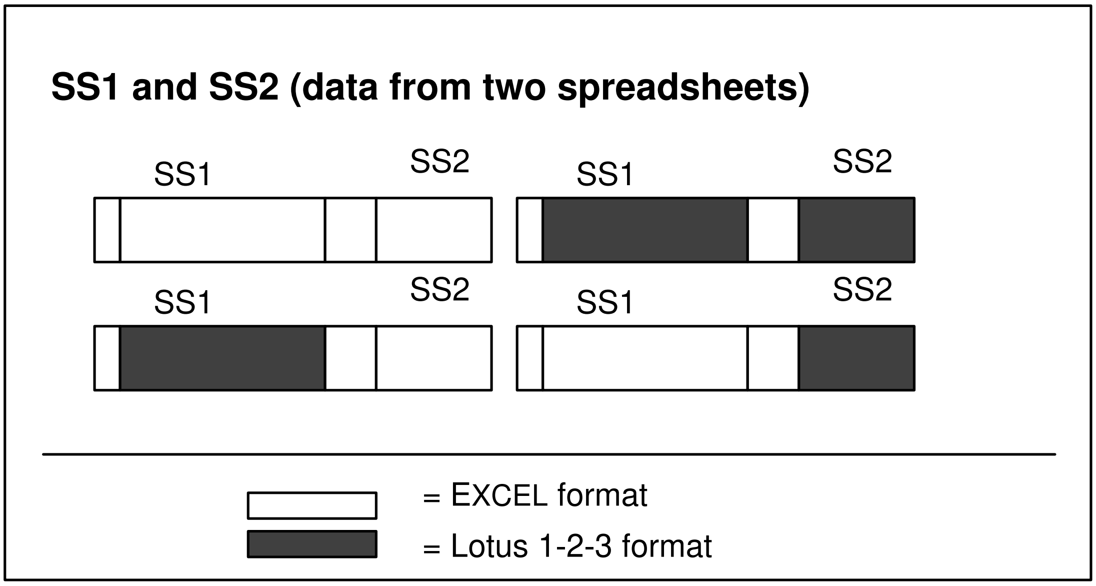
```
2 (4)name (5)Roger (5)phone (8)555-8585
```

Similarly, the remainder of the line gives the name and value of the second attribute.

Note: This description is conceptual. The actual storage format is not identical to the examples above.

Both the persistent object format and the I/O group format can be changed, but the procedure for changing the POF is quite different from the procedure for changing the IOGF. This section only concerns changing the POF. See Appendix D, "Subclassing the Persistence Framework," for more detail concerning changing the Persistence Framework.

One reason for changing an object's POF is to store it in a format compatible with some other object. For example, an EXCEL spreadsheet might be stored in a Lotus 1-2-3 format. Given two spreadsheets, any of the following POF formats might be used, as illustrated:



Creating a new persistent object format involves creating a new encoder/decoder class.

Encoder/Decoders

Every persistent object is associated with an *Encoder/Decoder.* An Encoder/Decoder is the <u>*class* that knows how to read and write the object's persistent object format</u>. Because a given persistent object can potentially be associated with many different Encoder/Decoders, the actual association between an object and an Encoder/Decoder is done at run time.

The Encoder/Decoder supplies two methods: a "Write" method, which converts a persistent object from its internal (run-time) form to a form suitable for storage on the persistent media, and a "Read" method, which performs the inverse conversion. The Write/Read methods must also preserve and recover any relationships (pointers) between the persistent objects.

The Persistence Framework supplies a default Encoder/Decoder class, **SOMPAttrEncoderDecoder**, which reads and writes the default POF described above. You can also create your own Encoder/Decoder classes for your persistent objects. When an application supplies its own Encoder/Decoder class, it is free to decide how data is passed between the Encoder/Decoder and the persistent object. If your persistent data must be maintained in private instance variables or is not easily represented in one of the IDL data types, you must build your own Encoder/Decoder class.

The default Encoder/Decoder

The supplied default Encoder/Decoder class, **SOMPAttrEncoderDecoder**, requires the client application to make each element of persistent data an attribute, and to signify each persistent attribute with the “persistent” modifier in the .idl file. The “phoneDir” and “dirEntry” example classes used throughout this chapter have used the default Encoder/Decoder.

The default Encoder/Decoder makes use of attribute “get” and “set” methods. Whenever you define a class with an attribute, the SOM Compiler defines “get” and “set” methods for that attribute and supplies a default implementation. The default implementations of the “set” methods do only a shallow copy of the data passed to them. For example, when you set a string attribute, only the pointer to the string of data is copied. The string itself is not copied and therefore the storage containing the string must not be freed by the caller of the “set” method.

When the default Encoder/Decoder restores your data, it allocates space to contain the data it reads. It then invokes the object’s “set” methods to copy the restored data into the object. The default Encoder/Decoder assumes you are using the supplied “set” methods that do a shallow copy of the data. Therefore, the Encoder/Decoder does not free any of the storage it has allocated.

If you use the supplied “set” methods in your client applications, you may unwittingly cause a memory leak. To avoid this potential memory leak, you can take control of the implementation of the “set” method by using the **noset** modifier for the attribute. For example, to write your own “set” method for the “name” attribute of the “dirEntry” class, you would first change the .idl file of the class to indicate that you want to implement the “set” method:

```
name: noset, persistent;
```

Then, implement the “set” method along with the remainder of the methods. For example:

```
SOM_Scope void  SOMLINK _set_name(dirEntry somSelf, string name)
{
     dirEntryData *somThis = dirEntryGetData(somSelf);
     dirEntryMethodDebug("dirEntry", "_set_name");

     if (_name) SOMFree(_name);
     _name = (string)SOMMalloc(strlen(name)+1);
     strcpy (_name, name);
}
```

Note: This assumes that the **somInit** method of the class has initialized the attribute variables. In the example, **somInit** should set the _name variable to NULL, so that the first invocation of _set_name works correctly.

Writing an Encoder/Decoder

An Encoder/Decoder for a persistent object is typically implemented by the same person who implemented the persistent object's class. To write an Encoder/Decoder, derive a new class from **SOMPEncoderDecoderAbstract** and override the **sompEDWrite** and **sompEDRead** methods, as shown below:

```
#include <eda.idl>
interface MyEncoderDecoder : SOMPEncoderDecoderAbstract
{
  #ifdef __SOMIDL__
  implementation
  {
     sompEDWrite: override;
     sompEDRead: override;
  };
  #endif /* __SOMIDL__ */
};
```

These two methods are prototyped in IDL as follows:

```
void sompEDWrite(in SOMPMediaInterfaceAbstract thisMedia,
                 in SOMPPersistentObject thisObject)
        raises (sompException);

void sompEDRead(in SOMPMediaInterfaceAbstract thisMedia,
                in SOMPPersistentOBject thisObject)
        raises (sompException);
```

Each of these methods takes (a) a pointer to a *Media Interface* object and (b) a pointer to the persistent object being read/written. A *Media Interface* object is an instance of a class derived from **SOMPMediaInterfaceAbstract**. As its name implies, it provides the interface between the Persistence Framework and the media onto which an object's data is stored. Its methods perform the tasks required for file input and output. The Encoder/Decoder uses the Media Interface to perform actual I/O to the persistent media. The Media Interface Class passed to an Encoder/Decoder is determined by the I/O Group Manager object that has been specified as part of a persistent object's ID. The supplied I/O Group Manager **SOMPAscii** uses the Media Interface **SOMPAsciiMediaInterface**. The **SOMPBinary** I/O Group Manager uses **SOMPBinaryFileMedia**.

Once a new Encoder/Decoder class has been implemented, the Persistence Framework must be told to use it, rather than the default Encoder/Decoder class. The Encoder/Decoder can be reset at either the object or the class level:

- At the object level, reset the encoder/decoder via the **SOMPPersistentObject** method **sompSetEncoderDecoderName**.
- At the class level, reset the encoder/decoder via the **M_SOMPPersistentObject** method **sompSetClassLevelEncoderDecoderName**. (The **M_SOMPPersistentObject** class is the metaclass of **SOMPPersistentObject** which can be returned by **somGetClass** on any persistent object).

Methods supporting encoder/decoders

A custom Encoder/Decoder is responsible for knowing how to store an object's persistent data via a given Media Interface object, and knowing how to restore an object's data via a Media Interface and restore it into the object. A set of methods defined in the **SOMPFileMediaAbstract** class are designed to make this easier. They are listed here in the IDL form used for defining each method. Also, all raise **sompException**, which is not shown here.

```
   void sompWriteBytes(in string byteStream, in long length)
// Write a byte stream of the given length to the media.

   void sompWriteOctet(in octet i1)
// Writes the given 8-bit integer

   void sompWriteShort(in short i2)
// Writes the given short integer

   void sompWriteUnsignedShort(in ushort u2)
// Writes the given unsigned short integer

   void sompWriteLong(in long i4)
// Writes the given long integer

   void sompWriteUnsignedLong(in ulong u4)
// Writes the given unsigned long integer

   void sompWriteDouble(in double f8)
// Writes the given double-precision float

   void sompWriteFloat(in float f4)
// Writes the given float

   void sompWriteCharacter(in char c)
// Writes the given character

   void sompWriteString(in string wstring)
// Writes the given string in the following format:
//    (string length)(wstring data)

   void sompWriteLine(in string buffer)
// Writes the given string in <buffer> at the current position.
// The terminating null character (\0) is not written.
//
// This method does NOT append a newline character (\n) to the
// given string before writing. If the caller intends to
// restore this <buffer> of data with sompReadLine, the user must
// put the newline character in the <buffer> before calling this
// method.


   void sompReadBytes(in string byteStream, in long length)
// Read a byte stream of the given length from the media.

   void sompReadOctet(inout octet i1)
// Reads an 8-bit integer into memory at given the pointer

   void sompReadShort(inout short i2)
// Reads a short integer into memory at given the pointer

   void sompReadUnsignedShort(inout unsigned short u2)
// Reads an unsigned short integer into memory at given the pointer

   void sompReadLong(inout long i4)
// Reads a long integer into memory at given the pointer

   void sompReadUnsignedLong(inout ulong i4)
// Reads a long integer into memory at given the pointer
```

```
   void sompReadDouble(inout double f8)
// Reads a float into memory at given the pointer

   void sompReadFloat(inout float f4)
// Reads a float into memory at given the pointer

   void sompReadCharacter(in string c)
// Reads a character into memory at given the pointer

   void sompReadString(inout string rstring)
// Read and allocate a string. Input is a pointer to a string.
// The input pointer is modified to point to a newly allocated
// buffer which will contain the string read from the media.
// Callers are responsible for freeing the returned buffer with
// SOMFree. If you want to read a string into a predefined buffer,
// use sompReadStringToBuffer.
//
// This method can be used to read strings which were stored by
// sompWriteString.   void sompReadSomid(inout somId *id)
// Reads the string on the media with sompReadString and converts
// it to a somId.

   void sompReadStringToBuffer(in string buffer, in long bufsize)
// Read a string into the preallocated buffer given. The size of the
// buffer is given in <bufsize>. If the string read is larger than
// bufsize it is truncated to fit in bufsize.
//
// This method can be used to read strings which were stored by
// sompWriteString.

   void sompReadLine(in string buffer, in long bufsize)
// Read a string up to and including the first newline character
// (\n) into the preallocated buffer given.
// Use this method for reading strings stored with sompWriteLine.
//
// The size of the buffer is given in <bufsize>. If the string read
// is larger than bufsize, it is truncated to fit in bufsize.
//
// The characters read are stored in <buffer>, and a null character
// (\0) is appended. The newline character, if read, is included in
// the string.
```

These methods are all designed to make it simple to read and write the basic IDL data types. Two methods merit special attention:

```
   void sompWriteSomobject(in SOMObject so,
                           in SOMObject parentObject)
// Writes the given object.  Uses parentObject to determine whether
// relative Ids are stored.  If object (so) has no persistent parent,
// client passes NULL parentObject.

   void sompReadSomobject(inout SOMObject so)
// Instantiates and returns a new object of the class specified in
// the file. Standard SOM objects are simply instantiated.
// Persistent objects are instantiated and marked for restoration.
```

The first of these methods is used to store an object. It is used in the **sompEDWrite** method of an Encoder/Decoder when, while storing one persistent object, a pointer to another (a child object) is encountered that should also be written. The **sompWriteSomobject** method adds the child object to the set of objects to be stored by the Persistence Framework with

sompAddObjectToWriteSet, and then writes the ID string of the child object. If the child object is nonpersistent, the only thing written is the name of the class; consequently, on restore, an instantiated object with no data will be returned. If the other object is persistent, it is stored. (The actual storage of the child object is deferred until the write of the current object is complete, thus reducing the number of files open concurrently.) The first argument to the method (following the Media Interface object and Environment) is the object to be written; the second is the parent of that object (typically the object being stored at the time the object was encountered).

The second of these methods is used to read in an object that was stored previously with **sompWriteSomobject**. There should be one call to **sompReadSomobject** for every **sompWriteSomobject** that was called when the object was stored. **sompReadSomobject** first restores the Persistent ID of the child object and then recursively uses **sompRestoreObject** on the Persistent Storage Manager to restore the child object.

Both of these methods behave differently if the client application has invoked either of the methods **sompStoreObjectWithoutChildren** or **sompRestoreObjectWithoutChildren**. **sompWriteSomobject** stores the ID of the object, but the object itself it not stored (unless it is in the same I/O Group as the object currently being stored). **sompReadSomobject** reads an object ID and creates a new object, but the new object's data is not read.

Example 7: Encoder/Decoder example implementation

In this example, two Encoder/Decoders are created. The Encoder/Decoder class for use with the "dirEntry" class is "entryED". For the "phoneDir" class, it is "dirED". Each Encoder/Decoder has two methods: **sompEDWrite**, which encodes a persistent object's data, and **sompEDRead**, which decodes the stored data and places it into the persistent object.

The "dirEntry" Encoder/Decoder — "entryED"

The definition of the "dirEntry" encoder/decoder is as follows:

```
#include <eda.idl>
interface entryED: SOMPEncoderDecoderAbstract{
   #ifdef __SOMIDL__
   implementation {
       sompEDWrite: override;
       sompEDRead: override;
    };
    #endif
};
```

Method "sompEDWrite": The **sompEDWrite** method for directory entries writes each element of instance data (name and phone) using the **sompWriteString** method available on its given Media Interface.

```
SOM_Scope void  SOMLINK sompEDWrite(entryED somSelf,
                         Environment *ev,
                         SOMPMediaInterfaceAbstract thisMedia,
                         SOMPPersistentObject thisObject)
{
        entryEDMethodDebug("entryED","sompEDWrite");

      /* Write name
       ----------- */
       _sompWriteString(thisMedia, ev, __get_name(thisObject));
       if (ev->_major != NO_EXCEPTION) return;

     /* Write phone number string
       ------------------------ */
       _sompWriteString(thisMedia, ev, __get_phone(thisObject));
       return;
}
```

Method "sompEDRead": The **sompEDRead** method for directory entries reads the strings written out by **sompWriteString** with the analogous read method **sompReadString**.

```
SOM_Scope void  SOMLINK sompEDRead(entryED somSelf,
                        Environment *ev,
                        SOMPMediaInterfaceAbstract thisMedia,
                        SOMPPersistentObject thisObject)
{
      string rstring;

      entryEDMethodDebug("entryED","sompEDRead");

   /* Read name
      rstring is malloc'd by sompReadString
      ------------------------------------ */
      _sompReadString(thisMedia, ev, &rstring);
      if (ev->_major != NO_EXCEPTION) return;
      __set_name(thisObject, rstring);

   /* Read phone number
      ----------------- */
      _sompReadString(thisMedia, ev, &rstring);
      if (ev->_major != NO_EXCEPTION) return;
      __set_phone(thisObject, rstring);
      return;
}
```

The "phoneDir" encoder/decoder — "dirED"

The Encoder/Decoder methods for the "phoneDir" object are somewhat more complex, since the instance data for a "phoneDir" object includes pointers to its "dirEntry" children. These child objects must be stored by the **sompEDWrite** method, and must be restored by **sompEDRead**.

The definition of the "phoneDir" Encoder/Decoder is as follows:

```
#include <eda.idl>
interface dirEd: SOMPEncoderDecoderAbstract{
   #ifdef __SOMIDL__
   implementation {
       sompEDWrite: override;
       sompEDRead: override;
    };
    #endif
};
```

Method sompEDWrite: The **sompEDWrite** method for "phoneDir" objects first writes the maximum and actual number of entries in the directory using the method **sompWriteLong**. After writing the number of entries, the **sompEDWrite** method then loops for the length of the sequence, writing out the contained persistent objects using the **sompWriteSomobject** method.

```
#define dirED_Class_Source
#include "dired.ih"

#include <somp.h>
#include <phonedir.ih>
#include <direntry.ih>
```

```
SOM_Scope void  SOMLINK sompEDWrite(dirED somSelf, Environment *ev,
                SOMPMediaInterfaceAbstract thisMedia,
                SOMPPersistentObject thisObject)
{
/* Local declarations.
   ------------------- */
   long dlen;
   int i;
   _IDL_SEQUENCE_dirEntry directory;

/* dirEDData *somThis = dirEDGetData(somSelf); */
   dirEDMethodDebug("dirED","sompEDWrite");

/* Write out sequence max
   ---------------------------- */
   directory = __get_directory(thisObject);
   _sompWriteLong(thisMedia, ev, directory._maximum);

/* Write out number of entries.
   ---------------------------- */
   dlen = directory._length;
   _sompWriteLong(thisMedia, ev, dlen);

/* Write out the persistent Ids for each child.
   -------------------------------------------- */
   for (i=0;(i < dlen) && (ev->_major == NO_EXCEPTION);i++) {
      _sompWriteSomobject(thisMedia, ev, directory._buffer[i],
      thisObject);
   } /* endfor */

   return;
}
```

Method "sompEDRead": The **sompEDRead** method is the mirror image of the **sompEDWrite** method. It first reads the maximum and actual number of entries in the directory sequence with **sompReadLong**. Then it restores the "dirEntry" objects into the directory sequence buffer with **sompReadSomobject**. Finally, it sets the "phoneDir" sequence with a call to set_directory.

```
SOM_Scope void  SOMLINK sompEDRead(dirED somSelf, Environment *ev,
                SOMPMediaInterfaceAbstract thisMedia,
                SOMPPersistentObject thisObject)
{
/* Local Declarations.
   ------------------- */
   int i;
   long dlen, dmax;
   SOMObject ro;
   _IDL_SEQUENCE_dirEntry directory;

   dirEDMethodDebug("dirED","sompEDRead");

/* Read maximum and actual size of sequence.
   ----------------------------------------  */
   _sompReadLong(thisMedia, ev, &dmax);
   directory._maximum = dmax;
   _sompReadLong(thisMedia, ev, &dlen);
   directory._length = dlen;

/* Allocate space for sequence entries.
   ------------------------------------ */
   if (ev->_major == NO_EXCEPTION) {
      directory._buffer = (dirEntry *)
        SOMMalloc(dmax*sizeof(dirEntry));

/* Restore the individual entries to the name space.
   ------------------------------------------------- */
   for (i=0;i < dlen && (ev->_major == NO_EXCEPTION);i++) {
      _sompReadSomobject(thisMedia, ev, (SOMObject*)(&ro));
      *(directory._buffer + i) = ro;
   } /* endfor */
   if (ev->_major == NO_EXCEPTION) {

/*    Copy structure into attribute.
      ------------------------------ */
      __set_directory(thisObject, &directory);
    }
  }

  return;
}
```

The definition and implementation of the "dirEntry" and "phoneDir" classes are unchanged from the previous example. The test program is unchanged except for small additions to the main program, which appears as follows (with significant differences shown in bold):

```
main()
{

/* Initialize persistent framework.
   -------------------------------- */
   psm            = SOMPPersistentStorageMgrNew();
   diredClass     = dirEDNewClass (0,0);
   entryedClass   = entryEDNewClass (0,0);
   ev             = somGetGlobalEnvironment();

...
```

In order for the Persistence Framework to instantiate a user-written Encoder/Decoder, its class object must exist. Because you wouldn't normally instantiate an Encoder/Decoder object on your own, the <*className*>**NewClass** procedure of your Encoder/Decoder classes must be called. The highlighted lines in the main program create the class objects for the "dirED" and "entryED" classes with an explicit call to their <*className*>**NewClass** procedures. These classes could also be built into a dynamically loadable class library where their **NewClass** procedures are called automatically. For more information on building dynamically loadable class libraries, see "Creating a SOM Class Library" in Chapter 5, "Implementing Classes in SOM."

Encoder/Decoder class objects must also exist at the time an object is restored.

The save function with modifications is shown below. Two methods are shown for setting an object's Encoder/Decoder. For the "phoneDir" object "mylist", the Encoder/Decoder class is set with the **sompSetEncoderDecoderName** method. This sets the Encoder/Decoder for only the "mylist" object and no others. **sompSetClassLevelEncoderDecoderName** is used to set the Encoder/Decoder for all objects of class "dirEntry". The restore function in the example does not have to set the name of an object's Encoder/Decoder class name because the class name is stored as part of an object's data and restored when the object is restored.

```
save()
{
/* Local declarations.
   ------------------- */
   SOMPPersistentId pid;
   dirEntry  name1, name2;
   phoneDir  mylist;
   string    fp;

/* Create the persistent objects.
   ------------------------------ */
   mylist = phoneDirNew();
   name1  = dirEntryNew();
   name2  = dirEntryNew();

/* Set the encoder/decoder class.
   This call sets for only "mylist" object.
   ---------------------------------------- */
   _sompSetEncoderDecoderName (mylist, ev, "dirED");

/* Set the encoder/decoder class.
   This call sets for all objects of class "dirEntry"
   -------------------------------------------------- */
   _sompSetClassLevelEncoderDecoderName (_somGetClass(name1), ev,
                                         "entryED");

/* Create the persistent Id.
   ------------------------- */
   pid = SOMPPersistentIdNew();
   _sompSetIOGroupName(pid, ev, "./phoneDir");
   if (ev->_major != NO_EXCEPTION) exit(-1);
   _sompInitGivenId (mylist, ev, pid);
   _somFree (pid);
   if (ev->_major != NO_EXCEPTION) exit(-1);

/* Get the object id used for storing the directory.
   ------------------------------------------------- */
   pid = _sompGetPersistentId (mylist, ev);
   fp  = _sompGetIOGroupName (pid, ev, idBuff);
```

```
/* Add entry 1.
   ------------ */
   _mkEntry  (name1, "Roger", "555-5085");
   _sompInitNearObject (name1, ev, mylist);
   if (ev->_major != NO_EXCEPTION) exit(-1);
   _addEntry (mylist, name1);

/* Add entry 2.
   ------------ */
   _mkEntry  (name2, "Robert", "555-8151");
   _sompInitNearObject (name2, ev, mylist);
   if (ev->_major != NO_EXCEPTION) exit(-1);
   _addEntry (mylist, name2);

/* Display phone directory.
   ------------------------ */
   _printDirInfo(mylist);

/* Store the phone directory.
   -------------------------- */
   _sompStoreObject (psm, ev, mylist);
   if (ev->_major != NO_EXCEPTION) {
      somPrintf ("\nBack from StoreObject - ERROR!\n");
   } else {
      somPrintf ("\nBack from StoreObject - Ok\n");
      somPrintf ("Group Name is %s\n", idBuff);
   }
/* Finished.
   --------- */
   _somFree (name2);
   _somFree (name1);
   _somFree (mylist);}
```

8.7 Multi-thread Considerations

The Persistence Framework allows multi-threaded applications on OS/2. Applications can safely have multiple threads, each doing stores or restores for different objects. The Persistence Framework does not guarantee that persistent objects are multi-thread enabled. This is the responsibility of the object implementor. In most cases, having more than a small number of threads doing store/restore concurrently is less efficient than allowing the store/restores to occur sequentially, since more time must be spent in physically seeking on the disk.

8.8 Error Handling

The Persistence Framework uses the CORBA specification for error handling. As specified by the CORBA standard, a method may return zero or more “exceptions”. CORBA “exceptions” are implemented by passing back error information after a method call, as opposed to the “catch/throw” model, in which an exception is implemented by a long jump or signal. Each defined exception has an error data structure whose value is accessible to the caller after calls to methods that use that exception. (For additional information, consult the CORBA 1.1 specification.)

All Persistence Framework methods that return errors do so by raising the **sompException** exception. The exception is returned via the **Environment** parameter. To illustrate how error handling is managed, we present a simple “animal” class that returns errors exactly like the Persistence Framework. The animal.idl file looks like this:

```
#include <somobj.idl>
#include <somperrd.idl>
interface animalErr: SOMObject
{
  void makeNoise()   raises (sompException);
  void getFood()     raises (sompException) ;
  void printAnimal() raises (sompException);
};
```

The **sompException** is defined as follows:

```
exception sompException {
  long primary;
  long secondary;
};
```

As with any CORBA exception, the client checks for an exception after a method call by examining the **_major** field of the **Environment** structure. This field is guaranteed to equal NO_EXCEPTION if no error has occurred. In general, when a method returns an error via the **Environment** structure, the **_major** field is set to either SYSTEM_EXCEPTION or USER_EXCEPTION. When an error occurs in the Persistence Framework, **_major** is always set to USER_EXCEPTION. A client program that makes use of the “animal” class is shown below:

```
main()
{ animalErr pooh = animalErrNew();
  dogErr snoopie = dogErrNew();
  Environment *myenv = SOM_CreateLocalEnvironment();
  void displayErr(Environment *ev);

  _makeNoise(pooh, myenv);
  if (myenv->_major != NO_EXCEPTION) displayErr(myenv);

  _getFood(pooh, myenv);
  if (myenv->_major!= NO_EXCEPTION) displayErr(myenv);

  _printAnimal(pooh, myenv);
  if (myenv->_major!= NO_EXCEPTION) displayErr(myenv);

  _getFood(snoopie, myenv);
  if (myenv->_major!= NO_EXCEPTION) displayErr(myenv);

  SOM_DestroyLocalEnvironment(myenv);
  return (int) 0;
}
```

Once the client determines that an exception has occurred, additional information can be retrieved via the exception members. The Persistence Framework exception, **sompException**, contains both a **_primary** and **_secondary** field, which further define the exception. The following “displayErr” function demonstrates how to use these fields:

```
void displayErr(Environment *ev)
{
  sompException *params;
  switch(ev->_major) {
  case SYSTEM_EXCEPTION:
    somPrintf(”System Exception Raised\n”);
    break;
  case USER_EXCEPTION:
    somPrintf(”User Exception Raised\n”);
    somPrintf(”ID of exception: %s\n”, somExceptionId(ev));
    params = somExceptionValue(ev);
    somPrintf(”  primary: %d\n”, params->primary);
    somPrintf(”secondary: %d\n”, params->secondary);
    break;
  default:
    somPrintf(”Unknown Exception Raised\n”);
    break;
  }
}
```

In general, the value returned from **somExceptionValue** is not the same for all USER_EXCEPTION errors. You must check the exception ID returned with **somExceptionId**. However, to check specifically for Persistence Framework errors, you could simplify the above function as shown:

```
void displayErr(Environment *ev)
{
  sompException *params;                    /* Declare parameters */
  somPrintf(”User Exception Raised\n”);     /* Print warning      */
  params = somExceptionValue(ev);          /* Get parameters     */
  somPrintf(”primary: %d\n”, params->primary);
                                        /* Display primary error */
  somPrintf(”secondary: %d\n”, params->secondary);
                                      /* Display secondary error */
}
```

Another way to manage exceptions would be to place error management code in the client program directly:

```
  _makeNoise(pooh, myenv);
  if (myenv->_major != NO_EXCEPTION){
      /* Error handling code. */
  }

  _getFood(pooh, myenv);
  if (myenv->_major!= NO_EXCEPTION){
     /* Error handling code. */
  }

  etc.
```

Error codes

The members of **sompException, primary** and **secondary** have a predefined set of possible values. The **primary** field will contain either the value SOMPERROR_SYSTEM_ERROR or SOMPERROR_FRAMEWORK_ERROR. The former is returned when the originating error comes from the underlying C library. The latter is returned when the error was detected within the Persistence Framework.

When the **primary** field is SOMPERROR_SYSTEM_ERROR, **secondary** will contain the actual error returned from the C library (the value normally found in the C variable “errno”).

When the **primary** field is SOMPERROR_FRAMEWORK_ERROR, **secondary** will contain one of the predefined errors, defined in “somperr.idl”.

For a listing and explanation of the Persistence Framework error codes, please refer to Appendix A, “Customer Support and Error Codes,” which lists the error codes for all frameworks of the SOMobjects Developer Toolkit.