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, "Implementing Classes in SOM.") 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** 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
```

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

The **sc** 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, "Implementing Classes in SOM"). To run the Interface Repository emitter by itself, issue the **sc** command with the –s option (which overrides SMEMIT) set to "ir". For example:

```
sc –u –sir newcls
```

or equivalently,

```
sc –usir newcls
```

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.

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
```

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
```

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.

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** command. For example, the command

```
sc –sh –mtcconsts myclass.idl
```

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. Utility Metaclasses in SOMobjects Toolkit

Contents

Chapter 8. Utility Metaclasses in SOMobjects Toolkit

8.1 Utility Metaclasses

In SOM, classes are objects; metaclasses are classes and thus are objects, too. Figure 1 depicts the relationship of these sets of objects. Also depicted are the three primitive class objects of the SOM run time: **SOMClass**, **SOMObject**, and **SOMClassMgr**.

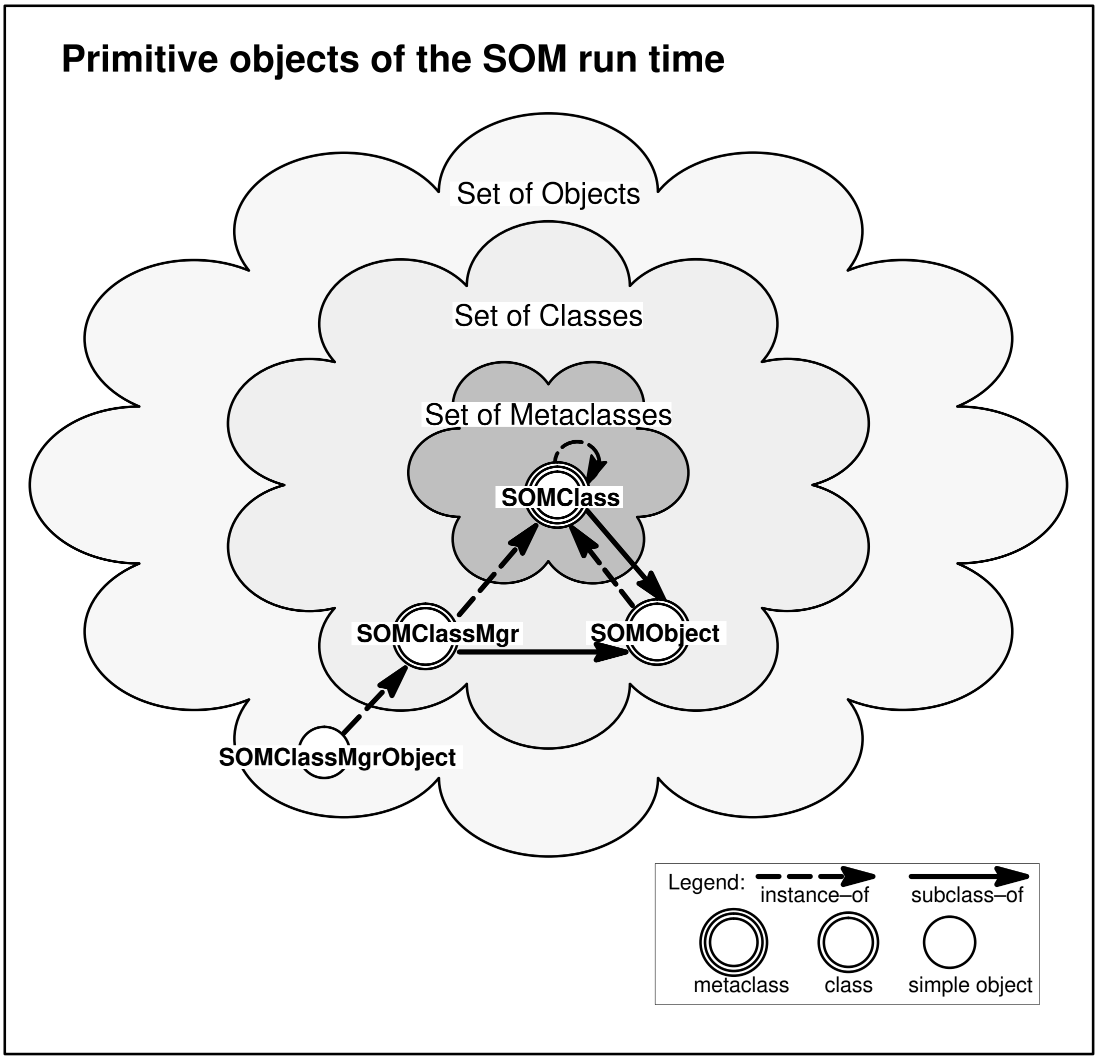

Figure 1. The primitive objects of the SOM run time.

The important point to be aware of here is that any class that is a *subclass* of **SOMClass** is a *metaclass* (because **SOMClass**'s object-creation method is inherited, and that method creates a class). The current section describes the **SOMMSingleInstance** metaclass that is available as part of the SOMobjects Developer Toolkit utilities. Figure 2 depicts the relationship of the **SOMMSingleInstance** metaclass to **SOMClass**:

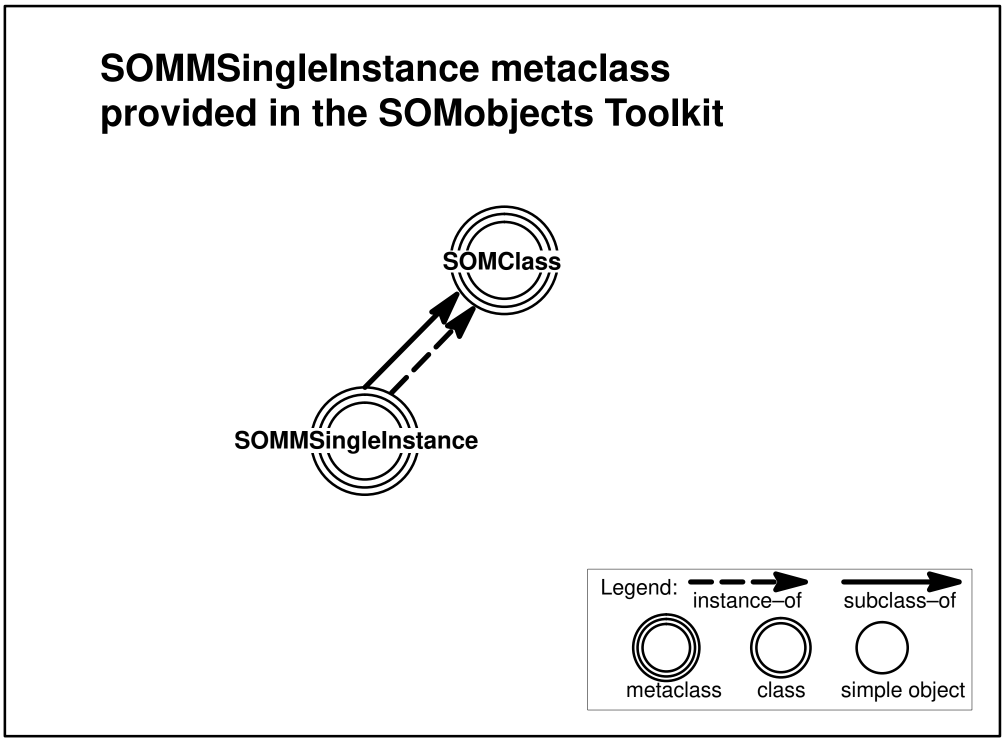

Figure 2. The SOMMSingleInstance metaclasses is provided in the SOMobjects Toolkit.

SOMMSingleInstance — Used to create a class that may have at most one instance.

The "SOMMSingleInstance" metaclass

Sometimes it is necessary to define a class for which only one instance can be created. This is easily accomplished with the **SOMMSingleInstance** metaclass. Suppose the class "Collie" is an instance of **SOMMSingleInstance**. The first call to **CollieNew** creates the one possible instance of "Collie"; hence, subsequent calls to **CollieNew** return the first (and only) instance.

Any class whose metaclass is **SOMMSingleInstance** gets the requisite behavior; nothing further needs to be done. The first instance created is always returned by the <*className*>**New** macro.

Alternatively, the *method* **sommGetSingleInstance** does the same thing as the <*className*>**New** macro. This method invoked on a class object (for example, "Collie") is useful because the call site explicitly shows that something special is occurring and that a new object is not necessarily being created. For this reason, you might prefer the second form of creating a single-instance object to the first.

Chapter 9. The Event Management Framework

Contents

Chapter 9. The Event Management Framework

The **Event Management Framework** is a central facility for registering all events of an application. Such a registration facilitates grouping of various application events and waiting on multiple events in a single event-processing loop. This facility is used by the DSOM Framework to wait on events of interest. The Event Management Framework must also be used by any interactive application that contains DSOM or replicated objects.

9.1 Event Management Basics

The Event Management Framework consists of an Event Manager (EMan) class, a Registration Data class and several Event classes. It provides a way to organize various application events into groups and to process all events in a single event-processing loop. The need for this kind of facility is seen very clearly in interactive applications that also need to process some background events (say, messages arriving from a remote process). Such applications must maintain contact with the user while responding to events coming from other sources.

One solution in a multi-threaded environment is to have a different thread service each different source of events. For a single-threaded environment it should be possible to recognize and process all events of interest in a single main loop. EMan offers precisely this capability. EMan can be useful even when multiple threads are available, because of its simple programming model. It avoids contention for common data objects between EMan event processing and other main-loop processing activity.

Model of EMan usage

The programming model of EMan is similar to that of many GUI toolkits. The main program initializes EMan and then registers interest in various types of events. The main program ends by calling a non-returning function of EMan that waits for events and dispatches them as and when they occur. In short, the model includes steps that:

1. Initialize the Event Manager,
2. Register with EMan for all events of interest, and
3. Hand over control to EMan to loop forever and to dispatch events.

The Event Manager is a SOM object and is an instance of the **SOMEEMan** class. Since any application requires only one instance of this object, the **SOMEEMan** class is an instance of the **SOMMSingleInstance** class. Creation and initialization of the Event Manager is accomplished by a function call to **SOMEEmanNew**.

Currently, EMan supports the four kinds of events described in the following topic. An application can register or unregister for events in a callback routine (explained below) even after control has been turned over to EMan.

Event types

Event types are categorized as follows:

- **Timer events**
  These can be either one-time timers or interval timers.
- **Sink events** (sockets, file descriptors, and message queues)
  On AIX, this includes file descriptors for input/output files, sockets, pipes, and message queues. On OS/2, only sockets are supported.

- **Client events** (any event that the application wants to queue with EMan)

  These events are defined, created, processed, and destroyed by the application. EMan simply acts as a place to queue these events for processing. EMan dispatches these client events whenever it sees them. Typically, this happens immediately after the event is queued.

- **Work procedure events** (procedures that can be called when there is no other event)

  These are typically background procedures that the application intends to execute when there are spare processor cycles. When there are no other events to process, EMan calls all registered work procedures.

The Event Management Framework is extendible (that is, other event types can be added to it) through subclassing. The event types currently supported by EMan are at a sufficiently low level so as to enable building other higher level application events on top of them. For example, you can build an X-event handler by simply registering the file descriptor for the X connection with EMan and getting notified when any X-event occurs.

Registration

This topic illustrates how to register for an event type.

Callbacks

The programmer decides what processing needs to be done when an event occurs and then places the appropriate code either in a procedure or in a method of an object. This procedure or method is called a *callback*. (The callback is provided to EMan at the time of registration and is called by EMan when a registered event occurs.) The signature of a callback is fixed by the framework and must have one of the following three signatures:

```
void EMRegProc(SOMEEvent, void *);
void SOMLINK EMMethodProc(SOMObject, SOMEEvent, void *);
void SOMLINK EMMethodProcEv(SOMObject, Environment *Ev,
                            SOMEEvent, void *);
/* On OS/2, they all use ”system” linkage  */
```

The three specified prototypes correspond to a simple callback procedure, a callback method using OIDL call style, and a callback method using IDL call style. The parameter type **SOMEEvent** refers to an event object passed by EMan to the callback. Event objects are described below.
Note: When the callbacks are methods, EMan calls these methods using **Name–lookup Resolution** (see Chapter 4, Section 4.3 on Method Resolution). One of the implications is that at the time of registration EMan queries the target object's class object to provide a method pointer for the method name supplied to it. Eman uses this pointer for making event callbacks.

Event classes

All event objects are instances of either the **SOMEEvent** class or a subclass of it. The hierarchy of event classes is as follows:

```
SOMObject ——— SOMEEvent ——— |————— SOMETimerEvent
                            |————— SOMEClientEvent
                            |————— SOMESinkEvent
                            |————— SOMEWorkProcEvent
```

When called by EMan, a callback expects the appropriate event instance as a parameter. For example, a callback registered for a timer event expects a **SOMETimerEvent** instance from EMan.

EMan parameters

Several method calls in the Event Management Framework make use of bit masks and constants as parameters (for example, **EMSinkEvent** or **EMInputReadMask**). These methods are defined in the include file “eventmsk.h”. When a user plans to extend the Event Management Framework, care must be taken to avoid name and value collisions with the definitions in “eventmsk.h”. For convenience, the contents of the “eventmsk.h” file are shown below.

```
#ifndef H_EVENTMASKDEF
#define H_EVENTMASKDEF

/* Event Types */
#define EMTimerEvent                    54
#define EMSignalEvent                   55
#define EMSinkEvent                     56

#define EMWorkProcEvent                 57

#define EMClientEvent                   58

#define EMMsgQEvent                     59

/* Sink input/output condition mask */

#define EMInputReadMask                 (1L<<0)
#define EMInputWriteMask                (1L<<1)
#define EMInputExceptMask               (1L<<2)

/* Process Event mask */

#define EMProcessTimerEvent             (1L<<0)
#define EMProcessSinkEvent              (1L<<1)
#define EMProcessWorkProcEvent          (1L<<2)
#define EMProcessClientEvent            (1L<<3)
#define EMProcessAllEvents              (1L<<6)

#endif   /* H_EVENTMASKDEF */
```

Registering for events

In addition to the event classes, the Event Management Framework uses a registration data class (**SOMEEMRegisterData**) to capture all event-related registration information. The procedure for registering interest in an event is as follows:

1. Create an instance of the **SOMEEMRegisterData** class (this will be referred to as a “RegData” object).
2. Set the event type of “RegData.”
3. Set the various fields of “RegData” to supply information about the particular event for which an interest is being registered.
4. Call the registration method of EMan, using “RegData” and the callback method information as parameters. The callback information varies, depending upon whether it is a simple procedure, a method called using OIDL call style, or a method called using IDL call style.

The following code segment illustrates how to register input interest in a socket “sock” and provide a callback procedure “ReadMsg”.

```
data = SOMEEMRegisterDataNew( );       /* create a RegData object */
_someClearRegData(data, Ev);
_someSetRegDataEventMask(data,Ev,EMSinkEvent,NULL); /* Event type */
_someSetRegDataSink(data, Ev, sock);     /* provide the socket id */
_someSetRegDataSinkMask(data,Ev, EMInputReadMask );
                                                 /*input interest */
regId = _someRegisterProc(some_gEMan,Ev,data,ReadMsg,"UserData" );
/* some_gEMan points to EMan. The last parameter "userData" is any
   data the user wants to be passed to the callback procedure as a
   second parameter */
```

Unregistering for events

One can unregister interest in a given event type at any time. To unregister, you must provide the registration id returned by EMan at the time of registration. Unregistering a non-existent event (such as, an invalid registration id) is a no-op. The following example unregisters the socket registered above:

```
_someUnRegister(some_gEMan, Ev, regId);
```

An example callback procedure

The following code segment illustrates how to write a callback procedure:

```
void SOMLINK ReadMsg( SOMEEvent  event, void  *targetData )
{
int sock;
    printf( "Data = %s\n", targetData );
    switch( _somevGetEventType( event )) {
    case  EMSinkEvent:
        printf("callback: Perceived Sink Event\n");
        sock = _somevGetEventSink(event);
        /* code to read the message off the socket */
        break;
    default: printf("Unknown Event type in socket callback\n");
    }
}
```

Generating client events

While the other events are caused by the operating system (for example, Timer), by I/O devices, or by external processes, client events are caused by the application itself. The application creates these events and enqueues them with EMan. When client events are dispatched, they are processed in a callback routine just like any other event. The following code segment illustrates how to create and enqueue client events.

```
clientEvent1 = SOMEClientEventNew();  /* create a client event */
_somevSetEventClientType( clientEvent1, Ev, "MyClientType" );
_somevSetEventClientData( clientEvent1, Ev,
                                   "I can give any data here");
/* assuming that "MyClientType" is already registered with EMan */
/* enqueue the above event with EMan */
_someQueueEvent(some_gEMan, Ev, clientEvent1);
```

Examples of using other events

The sample program shipped with the Event Management Framework illustrates the tasks listed below. (Due to its large size, the source code is not included here.)

- Registering and unregistering for Timer events.

- Registering and unregistering for Workproc events.
- Registering an AIX Message Queue, sending messages on it, and unregistering the Message Queue.
- Registering a stream socket that listens to incoming connection requests. Also, sockets connecting, accepting a connection, and sending/receiving messages through EMan.
- Registering a file descriptor on AIX and reading one line of the file at a time in a callback.

Processing events

After all registrations are finished, an application typically turns over control to EMan and is completely event driven thereafter. Typically, an application main program ends with the following call to EMan:

```
_someProcessEvents(some_gEMan, Ev);
```

An equivalent way to process events is to write a main loop and call **someProcessEvent** from inside the main loop, as indicated:

```
while (1) { /* do forever */
       _someProcessEvent( some_gEMan, Ev, EMProcessTimerEvent  |
                                          EMProcessSinkEvent    |
                                          EMProcessClientEvent  |
                                          EMProcessWorkProcEvent );
       /***  Do other main loop work, as needed. ***/
   }
```

The second way allows more precise control over what type of events to process in each call. The example above enables all four types to be processed. The required subset is formed by logically OR'ing the appropriate bit constants (these are defined in "eventmsk.h)". Another difference is that the second way is a non-blocking call to EMan. That is, if there are no events to process, control returns to the main loop immediately, whereas **someProcessEvents** is a non-returning blocking call. For most applications, the first way of calling EMan is better, since it does not waste processor cycles when there are no events to process.

Interactive applications

Interactive applications need special attention when coupled with EMan. Once control is turned over to EMan by calling **someProcessEvents**, a single-threaded application (for example, on AIX) has no way of responding to keyboard input. The user must register interest in "stdin" with EMan and provide a callback function that handles keyboard input. In a multi-threaded environment (for example, OS/2), this problem can be solved by spawning a thread to execute **someProcessEvents** and another to handle keyboard input. (These two options are illustrated in the sample program shipped with the Event Management Framework.)

9.2 Event Manager Advanced Topics

Threads and thread safety

As indicated earlier, on OS/2, interactive programs call **someProcessEvents** in one thread and process keyboard input in a separate thread. (This recommended usage is illustrated in the sample program). The event manager object (EMan) is thread safe in the sense that concurrent method invocations on EMan are serialized. Even when **someProcessEvents** is invoked in a thread and other methods of EMan are invoked from other threads, EMan still preserves its data integrity. However, when Eman dispatches an event, a callback can call methods on the same data objects as the other interactive thread(s). The user must protect such data objects using appropriate concurrency control techniques (for example by using semaphores).

One must also be aware of some deadlock possibilities. Consider the following situation. EMan code holds some SOMobjects Toolkit semaphores while it is running (for example, while in **someProcessEvents**). A user-defined object protects its data by requiring its methods to acquire and release a sempahore on the object. If a separate thread running in this object were to call an operation that requires a SOMobjects Toolkit semaphore (which is currently held by EMan) and if concurrently EMan dispatches an event whose callback invokes a method of this object, a deadlock occurs. Two possibilities exist to cope with such a situation: One is to acquire all needed semaphores ahead of time, and the other is to abort the operation when you fail to obtain a semaphore. To achieve mutual exclusion with EMan, you can call the methods **someGetEManSem** and **someReleaseEmanSem**. These methods acquire and release the SOMobject Developer Toolkit semaphores that EMan uses.

Writing an X or MOTIF application

Although the Event Manager does not recognize X events, an X or MOTIF application can be integrated with EMan as follows. First, the necessary initialization of X or MOTIF should be performed. Next, using the Xlib macro “ConnectionNumber” or the “XConnectionNumber” function, you can obtain the file descriptor of the X connection. This file descriptor can be registered with EMan as a sink. It can be registered for both input events and exception events. When there is any activity on this X file descriptor, the developer-provided callback is invoked. The callback can receive the X-event, analyze it, and do further dispatching.

Extending EMan

The current event manager can be extended without having access to the source code. The use of EMan in an X or MOTIF application mentioned above is just one such example. Several other extensions are possible. For example, new event types can be defined by subclassing either directly from **SOMEEvent** class or from any of its subclasses in the framework. There are three main problems to solve in adding a new event type:

- How to register a new event type with EMan?
- How to make EMan recognize the occurrence of the new event?
- How to make EMan create and send the new event object (a subclass of **SOMEEvent**) to the callback when the event is dispatched?

Because the registration information is supplied with appropriate “set” methods of a RegData object, the RegData object should be extended to include additional methods. This can be achieved by subclassing from **SOMERegisterData** and building a new registration data class that has methods to “set” and “get” additional fields of information that are needed to describe the new event types fully. To handle registrations with instances of new registration data subclass, we must also subclass from **SOMEEMan** and override the **someRegister** and the **someUnRegister** methods. These methods should handle the information in the new fields introduced by the new registration data class and call parent methods to handle the rest.

Making EMan recognize the occurrence of the new event is primarily limited by the primitive events EMan can wait on. Thus the new event would have to be wrapped in a primitive event that EMan can recognize. For example, to wait on a message queue on OS/2 concurrently with other EMan events, a separate thread can be made to wait on the message queue and to enqueue a client event with EMan when a message arrives on this message queue. We can thus bring multiple event sources into the single EMan main loop.

The third problem of creating new event objects unknown to EMan can be easily done by applying the previous technique of wrapping the new event in terms of a known event. In a callback routine of the known event, we can create and dispatch the new event unknown to EMan. Of course, this does introduce an intermediate callback routine which would not be needed if EMan directly understood the new event type.

A general way of extending EMan is to look for newly defined event types by overriding **someProcessEvent** and **someProcessEvents** in a subclass of EMan.

Using EMan from C++

The Event Management framework can be used from C++ just like any other framework in the SOMobjects Toolkit. You must ensure that the C++ usage bindings (that is, the .xh files) are available for the Event Management Framework classes. These .xh files are generated by the SOM Compiler in the SOMobjects Toolkit when the **-s** option includes an xh emitter.

Using EMan from other languages

The event manager and the other classes can be used from other languages, provided usage bindings are available for them. These usage bindings are produced from .idl files of the framework classes by the appropriate language emitter.

Tips on using EMan

The following are some do's and don'ts for EMan:

- Eman callback procedures or methods must return quickly. You cannot wait for long periods of time to return from the callbacks. If such long delays occur, then the application may not notice some subsequent events in time to process them meaningfully (for example, a timer event may not be noticed until long after it occurred).

- It follows from the previous tip that you should not do independent "select" system calls on file descriptors while inside a callback. (This applies to sockets and message queues, as well.) In general, a callback should not do any blocking of system calls. If an application must do this, then it must be done with a small timeout value.

- Since EMan callbacks must return quickly, no callback should wait on a semaphore indefinitely. If a callback has to obtain some semaphores during its processing, then the callback should try to acquire all of them at the very beginning, and should be prepared to abort and return to EMan if it fails to acquire the necessary semaphores.

- EMan callback methods are called using name-lookup resolution. Therefore, the parameters to an EMan registration call must be such that the class object of the object parameter must be able to provide a pointer to the method indicated by the method parameter. Although this requirement is satisfied in a majority of cases, there are exceptions. For example, if the object is a proxy (in the DSOM sense) to a remote object, then the "real" class object cannot provide a meaningful method pointer. Also note that, when **somDispatch** is overridden, the effect of such an override will *not* apply to the callback from EMan. Do not use a method callback in these situations; instead, use a procedure callback.

9.3 Limitations

The present implementation of the Event Management framework has the limitations described below. For a more up-to-date list of limitations, refer to the README file on EMan in the SOMobjects Developer Toolkit.

- EMan supports registering a maximum of 64 AIX message queues.
- EMan can only wait on file descriptors (including files, pipes, sockets, and message queues) on AIX, and socket identifiers on OS/2 and Windows.
- EMan supports registering a maximum of FILENO (the AIX limit on maximum number of open files) file descriptors on AIX. On OS/2 and Windows, the maximum number of socket identifiers depends on the underlying Sockets class.

Use of EMan DLL

The Event Manager Framework uses a **Sockets** "select" call to wait on multiple sockets. . At the time of EMan creation, the **SOMEEMan** class object loads one of the **Sockets** subclass DLLs, based on the value of the environment variable SOMSOCKETS. This environment variable should name the implementation class of sockets (see Appendix E describing the **Sockets** abstract class and the specific implementation DLLs available with the SOMobjects Toolkit.) The current choices for this environment variable are **TCPIPSockets**, **NBSockets**, and **IPXSockets**.

Appendix A. Customer Support and Error Codes

Service and Technical Support for SOMobjects

This service and technical support information applies for:

- **SOMobjects Developer Toolkit, Version 2.0**
- **SOMobjects Workstation Enabler, Version 2.0**
- **SOMobjects Workgroup Enabler, Version 2.0**

Notes: Customers in European, Middle Eastern, and African Countries should refer to the separate Service Statement included with the product for service and technical support instructions for this product.

Customers in Canada and Asia Pacific Countries should refer to the Service Statement in the License Information Booklet for service and technical support instructions for this product.

You Must Register for Service

Defect service for this product is available through September 30, 1995, or six months after the general availability of a subsequent version of the product (or a product designated as a replacement product), whichever occurs earlier.

Register by providing your company name, address, phone number, contact person's name, phone and FAX numbers (include area code). This information can be sent via electronic mail as follows:

- IBM OS2BBS to userid: **WZ00178**

or

- Internet Commercial: **somreg@austin.ibm.com**

or

- CompuServe: **GO IBMSOM**
  and then browse the News Flash for further registration information.

Within two working days of receipt of your registration, a service ID or password will be issued to you, allowing access to the defect forum and technical support forum.

Defect Support

Defect service for this product is available through September 30, 1995, or six months after the general availability of a subsequent version of the product (or a product designated as a replacement product), whichever occurs earlier.

Defect service is provided by the IBM SOMobjects Development personnel via the following Electronic Support Services:

- IBM OS/2 Bulletin Board System
  via IBM TalkLink Electronic Conferencing Service
- Internet Commercial Electronic Network
- CompuServe

The IBM SOMobjects Development personnel will monitor these Electronic Support Services between 8 a.m. and 7 p.m. CST, Monday through Friday, except holidays. Acknowledgement of receipt of Defect Report will be within 24 hours for SOMobjects RUNTIME defects and 72 hours for SOMobjects TOOLKIT defects, provided that the Defect Report is received by the SOMobjects Technical Support personnel during the time period of 8 a.m. to 7 p.m. CST, Monday through Friday.

Technical Support

Technical support service for this product is available for ninety (90) days after receipt of your service registration by SOMobjects Development personnel or until expiration of defect support, whichever occurs first.

Technical support service is provided by the IBM SOMobjects Development personnel via the following Electronic Support Services:

- IBM OS/2 Bulletin Board System
  via IBM TalkLink Electronic Conferencing Service
- Internet Commercial Electronic Network
- CompuServe

The IBM SOMobjects Development personnel will monitor these Electronic Support Services between 8 a.m. and 7 p.m. CST, Monday through Friday, except holidays. Questions will be answered in the order in which they are received. Extension of the technical support beyond the expiration date will be offered on a fee basis. Information regarding this offering will be provided on the service bulletin boards.

IBM OS/2 Bulletin Board System via TalkLink

The OS/2 Bulletin Board System (BBS) is implemented on the IBMLink facility. The OS/2 BBS is provided to all Workstation Technical Coordinators (WTSC) in corporate IBMLink accounts and all members of the OS/2 Developer's Assistance Program (DAP) who have access to IBMLink. You may contact your Technical Coordinator, if one has been identified by your company. If your company does not currently utilize IBMLink, you can subscribe to TalkLink by calling 1–800–547–1283 (USA).

How to use the IBM OS/2 Bulletin Board System (OS2BBS) via TalkLink for service and support for SOMobjects:

- To obtain **technical support** for non-defect "how-to" questions and answers:

  – Logon to IBM OS2BBS system from IBMLink Main Menu screen
  – Select "OS/2 Questions and Answer Bulletin Boards"
  – Select "SOMHOWTO" CFORUM

- To submit a suspected **defect report**:

  – Logon to IBM OS2BBS system from IBMLink Main Menu screen
  – Select "OS/2 Questions and Answer Bulletin Boards"
  – Select "SOMTKBUG" – if the suspected defect is with the SOM Toolkit
  – Select "SOMRTBUG" – if the suspected defect is with the SOM Runtime

Note: When submitting a suspected defect report, please provide the following information:

- Your Company name and address.
- Your name, phone and FAX numbers.
- The hardware platform – (PS/2 Model ____, or RS/6000 Model ____).
- Operating System and level – (OS/2 Version ____, or AIX Version ____).
- System configuration (memory, communication protocol, etc.).
- SOMobjects Version _____.
- Complete description of the problem.

Internet Commercial Electronic Network

How to use Internet for service and support for SOMobjects:

- To obtain **technical support**, for non-defect "how-to" questions and answers:
  - Via USENET Newsgroup at: **comp.unix.aix**
    Note: Include the word "SOM" in the subject line.
- To submit a suspected **defect report**:
  - Send EMAIL to: **sombug@austin.ibm.com**

Note: When submitting a suspected defect report, please provide the following information:

- Your Company name and address.
- Your name, phone and FAX numbers.
- The hardware platform (PS/2 Model ____, or RS/6000 Model ____).
- Operating System and level (OS/2 Version ____, or AIX Version ____).
- System configuration (memory, communication protocol, etc.).
- SOMobjects Version _____.
- Complete description of the problem.

CompuServe

How to use CompuServe for service and support for SOMobjects:

- From any CompuServe prompt, enter: **GO IBMSOM**

Note: When submitting a suspected defect report, please provide the following information:

- Your Company name and address.
- Your name, phone and FAX numbers.
- The hardware platform (PS/2 Model ____, or RS/6000 Model ____).
- Operating System and level (OS/2 Version ____, or AIX Version ____).
- System configuration (memory, communication protocol, etc.).
- SOMobjects Version _____.
- Complete description of the problem.

If you are not currently a member of CompuServe, you can subscribe by calling (USA) 1–800–524–3388 and asking for Representative 239.

SOM Kernel Error Codes

Following are error codes with messages/explanations for the SOM kernel and the various frameworks of the SOMobjects Developer Toolkit.

Value	Symbolic Name and Description
20011	`SOMERROR_CCNullClass` The **somDescendedFrom** method was passed a null class argument.
20029	`SOMERROR_SompntOverflow` The internal buffer used in **somPrintf** overflowed.
20039	`SOMERROR_MethodNotFound` **somFindMethodOk** failed to find the indicated method.
20049	`SOMERROR_StaticMethodTableOverflow` A Method–table overflow occurred in **somAddStaticMethod**.
20059	`SOMERROR_DefaultMethod` The **somDefaultMethod** was called; a defined method probably was not added before it was invoked.
20069	`SOMERROR_MissingMethod` The specified method was not defined on the target object.
20079	`SOMERROR_BadVersion` An attempt to load, create, or use a version of a class-object implementation is incompatible with the using program.
20089	`SOMERROR_NullId` The **SOM_CheckId** was given a null ID to check.
20099	`SOMERROR_OutOfMemory` Memory is exhausted.
20109	`SOMERROR_TestObjectFailure` The **somObjectTest** found problems with the object it was testing.
20119	`SOMERROR_FailedTest` The **somTest** detected a failure; generated only by test code.
20121	`SOMERROR_ClassNotFound` The **somFindClass** could not find the requested class.
20131	`SOMERROR_OldMethod` An old-style method name was used; change to an appropriate name.
20149	`SOMERROR_CouldNotStartup` The **somEnvironmentNew** failed to complete.
20159	`SOMERROR_NotRegistered` The **somUnloadClassFile** argument was not a registered class.
20169	`SOMERROR_BadOverride` The **somOverrideSMethod** was invoked for a method that was not defined in a parent class.
20179	`SOMERROR_NotImplementedYet` The method raising the error message is not implemented yet.
20189	`SOMERROR_MustOverride` The method raising the error message should have been overridden.

20199 **`SOMERROR_BadArgument`**
An argument to a core SOM method failed a validity test.

20219 **`SOMERROR_NoParentClass`**
During the creation of a class object, the parent class could not be found.

20229 **`SOMERROR_NoMetaClass`**
During the creation of a class object, the metaclass object could not be found.

DSOM Error Codes

The following table lists the error codes that may be encountered when using DSOM.

Value	Description
30001	**SOMDERROR_NoMemory** Memory is exhausted.
30002	**SOMDERROR_NotImplemented** Function or method has a null implementation.
30003	**SOMDERROR_UnexpectedNULL** Internal error: a pointer variable was found to be NULL, unexpectedly.
30004	**SOMDERROR_IO** I/O error while accessing a file located in SOMDDIR.
30005	**SOMDERROR_BadVersion** Internal error: incorrect version of an object reference data table.
30006	**SOMDERROR_ParmSize** Internal error: a parameter of incorrect size was detected.
30007	**SOMDERROR_HostName** Communications error: unable to retrieve local host name.
30008	**SOMDERROR_HostAddress** Communications error: unable to retrieve local host address.
30009	**SOMDERROR_SocketCreate** Communications error: unable to create socket.
30010	**SOMDERROR_SocketBind** Communications error: unable to bind address to socket.
30011	**SOMDERROR_SocketName** Communications error: unable to query socket information.
30012	**SOMDERROR_SocketReceive** Communications error: unable to receive message from socket.
30013	**SOMDERROR_SocketSend** Communications error: unable to send message to socket.
30014	**SOMDERROR_SocketIoctl** Communications error: unable to set socket blocking state.
30015	**SOMDERROR_SocketSelect** Communications error: unable to select on socket.
30016	**SOMDERROR_PacketSequence** Communications error: unexpected message packet received.
30017	**SOMDERROR_PacketTooBig** Communications error: packet too big for allocated message space.
30018	**SOMDERROR_AddressNotFound** Uninitialized DSOM communications object.
30019	**SOMDERROR_NoMessages** No messages available (and caller specified “no wait”).
30020	**SOMDERROR_UnknownAddress** Invalid client or server address.
30021	**SOMDERROR_RecvError** Communications error during receive.
30022	**SOMDERROR_SendError** Communications error during send.
30023	**SOMDERROR_CommTimeOut** Communications timeout.

30024	**SOMDERROR_CannotConnect** Unable to initialize connection information.
30025	**SOMDERROR_BadConnection** Invalid connection information detected.
30026	**SOMDERROR_NoHostName** Unable to get host name.
30027	**SOMDERROR_BadBinding** Invalid server location information in proxy object.
30028	**SOMDERROR_BadMethodName** Invalid method name in request message.
30029	**SOMDERROR_BadEnvironment** Invalid Environment value in request message.
30030	**SOMDERROR_BadContext** Invalid Context object in request message.
30031	**SOMDERROR_BadNVList** Invalid Named Value List (NVList).
30032	**SOMDERROR_BadFlag** Bad flag in NVList item.
30033	**SOMDERROR_BadLength** Bad length in NVList item.
30034	**SOMDERROR_BadObjref** Invalid object reference.
30035	**SOMDERROR_NullField** Unexpected null field in request message.
30036	**SOMDERROR_UnknownReposId** Attempt to use Invalid Interface Repository ID.
30037	**SOMDERROR_NVListAccess** Invalid NVList object in request message.
30038	**SOMDERROR_NVIndexError** Attempt to use an out-of-range NVList index.
30039	**SOMDERROR_SysTime** Error retrieving system time.
30040	**SOMDERROR_SystemCallFailed** System call failed.
30041	**SOMDERROR_CouldNotStartProcess** Unable to start a new process.
30042	**SOMDERROR_NoServerClass** No SOMDServer (sub)class specified for server implementation.
30043	**SOMDERROR_NoSOMDInit** Missing SOMD_Init call in program.
30044	**SOMDERROR_SOMDDIRNotSet** SOMDDIR environment variable not set.
30045	**SOMDERROR_NoImplDatabase** Could not open Implementation Repository database.
30046	**SOMDERROR_ImplNotFound** Implementation not found in implementation repository.
30047	**SOMDERROR_ClassNotFound** Class not found in implementation repository.
30048	**SOMDERROR_ServerNotFound** Server not found in **somdd**'s active server table.
30049	**SOMDERROR_ServerAlreadyExists** Server already exists in **somdd**'s active server table.

30050 **SOMDERROR_ServerNotActive**
Server is not active.

30051 **SOMDERROR_CouldNotStartSOM**
SOM initialization error.

30052 **SOMDERROR_ObjectNotFound**
Could not find desired object.

30053 **SOMDERROR_NoParentClass**
Unable to find / load parent class during proxy class creation.

30054 **SOMDERROR_DispatchError**
Unable to dispatch method.

30055 **SOMDERROR_BadTypeCode**
Invalid type code.

30056 **SOMDERROR_BadDescriptor**
Invalid method descriptor.

30057 **SOMDERROR_BadResultType**
Invalid method result type.

30058 **SOMDERROR_KeyInUse**
Internal object key is in use.

30059 **SOMDERROR_KeyNotFound**
Internal object key not found.

30060 **SOMDERROR_CtxInvalidPropName**
Illegal context property name.

30061 **SOMDERROR_CtxNoPropFound**
Could not find property name in context.

30062 **SOMDERROR_CtxStartScopeNotFound**
Could not find specified context start scope.

30063 **SOMDERROR_CtxAccess**
Error accessing context object.

30064 **SOMDERROR_CouldNotStartThread**
System error: Could not start thread.

30065 **SOMDERROR_AccessDenied**
System error: Access to a system resource (file, queue, shared memory, etc.) denied.

30066 **SOMDERROR_BadParm**
System error: invalid parameter supplied to a operating system call.

30067 **SOMDERROR_Interrupt**
System error: Interrupted system call.

30068 **SOMDERROR_Locked**
System error: Drive locked by another process.

30069 **SOMDERROR_Pointer**
System error: Invalid physical address.

30070 **SOMDERROR_Boundary**
OS/2 system error: ERROR_CROSSES_OBJECT_BOUNDARY.

30071 **SOMDERROR_UnknownError**
System error: Unknown error on operating system call.

30072 **SOMDERROR_NoSpace**
System error: No space left on device.

30073 **SOMDERROR_DuplicateQueue**
System error: Duplicate queue name.

30074 **SOMDERROR_BadQueueName**
System error: Invalid queue name.

30075 **SOMDERROR_DuplicateSem**
System error: Duplicate semaphore name used.

30076 **SOMDERROR_BadSemName**
System error: Invalid semaphore name.

30077 **SOMDERROR_TooManyHandles**
System error: Too many files open (no file handles left).

30078 **SOMDERROR_BadAddrFamily**
System error: Invalid address family.

30079 **SOMDERROR_BadFormat**
System error: Invalid format.

30080 **SOMDERROR_BadDrive**
System error: Invalid drive.

30081 **SOMDERROR_SharingViolation**
System error: Sharing violation.

30082 **SOMDERROR_BadExeSignature**
System error: Program file contains a DOS mode program or invalid program.

30083 **SOMDERROR_BadExe**
Executable file is invalid (linker errors occurred when program file was created).

30084 **SOMDERROR_Busy**
System error: Segment is busy.

30085 **SOMDERROR_BadThread**
System error: Invalid thread id.

30086 **SOMDERROR_SOMDPORTNotDefined**
SOMDPORT not defined.

30087 **SOMDERROR_ResourceExists**
System resource (file, queue, shared memory segment,. etc.) already exists.

30088 **SOMDERROR_UserName**
USER environment variable is not set.

30089 **SOMDERROR_WrongRefType**
Operation attempted on an object reference is incompatible with the reference type.

30090 **SOMDERROR_MustOverride**
This method has no default implementation and must be overridden.

30091 **SOMDERROR_NoSocketsClass**
Could not find/load Sockets class.

30092 **SOMDERROR_EManRegData**
Unable to register DSOM events with the Event Manager.

30093 **SOMDERROR_NoRemoteComm**
Remote communications is disabled (for Workstation DSOM).

Appendix B. SOM IDL Language Grammar

```
specification        : [comment] definition+

definition           : type_dcl  ; [comment]
                     | const_dcl ; [comment]
                     | interface ; [comment]
                     | module    ; [comment]
                     | pragma_stm

module               : module identifier [comment]
                       { [comment] definition+ }

interface            : interface identifier
                     | interface_dcl

interface_dcl        : interface identifier [inheritance] [comment]
                       { [comment] export* } [comment]

inheritance          : : scoped_name {, scoped_name}*

export               : type_dcl ; [comment]
                     | const_dcl ; [comment]
                     | attr_dcl ; [comment]
                     | op_dcl ; [comment]
                     | implementation_body ; [comment]
                     | pragma_stm

scoped_name          : identifier
                     | :: identifer
                     | scoped_name :: identifer

const_dcl            : const const_type identifier = const_expr

const_type           : integer_type
                     | char_type
                     | boolean_type
                     | floating_pt_type
                     | string_type
                     | scoped_name

const_expr           : or_expr

or_expr              : xor_expr
                     | or_expr | xor_expr

xor_expr             : and_expr
                     | xor_expr ^ and_expr

and_expr             : shift_expr
                     | and_expr & shift_expr

shift_expr           : add_expr
                     | shift_expr >> add_expr
                     | shift_expr <<  add_expr
```

```
add_expr            : mult_expr
                    | add_expr + mult_expr
                    | add_expr - mult_expr

mult_expr           : unary_expr
                    | mult_expr * unary_expr
                    | mult_expr / unary_expr
                    | mult_expr % unary_expr

unary_expr          : unary_operator primary_expr
                    | primary_expr

unary_operator      : -
                    | +
                    | ~

primary_expr        : scoped_name
                    | literal
                    | ( const_expr )

literal             : integer_literal
                    | string_literal
                    | character_literal
                    | floating_pt_literal
                    | boolean_literal

type_dcl            : typedef type_declarator
                    | constr_type_spec

type_declarator     : type_spec declarator {, declarator}*

type_spec           : simple_type_spec
                    | constr_type_spec

simple_type_spec    : base_type_spec
                    | template_type_spec
                    | scoped_name

base_type_spec      : floating_pt_type
                    | integer_type
                    | char_type
                    | boolean_type
                    | octet_type
                    | any_type
                    | voidptr_type

template_type_spec  : sequence_type
                    | string_type

constr_type_spec    : struct_type
                    | union_type
                    | enum_type

declarator          : [stars] std_declarator

std_declarator      : simple_declarator
                    | complex_declarator

simple_declarator   : identifier

complex_declarator  : array_declarator
```

```
array_declarator     : simple_declarator fixed_array_size+

fixed_array_size     : [ const_expr ]

floating_pt_type     : float
                     | double

integer_type         : signed_int
                     | unsigned_int

signed_int           : long
                     | short

unsigned_int         : unsigned signed_int

char_type            : char

boolean_type         : boolean

octet_type           : octet

any_type             : any

voidptr_type         : void stars

struct_type          : (struct|exception) identifier
                     | (struct|exception) [comment]
                      { [comment] member* }

member               : type_declarator ; [comment]

union_type           : union identifier
                     | union identifier switch
                       ( switch_type_spec ) [comment]
                       { [comment] case+ }

switch_type_spec     : integer_type
                     | char_type
                     | boolean_type
                     | enum_type
                     | scoped_name

case                 : case_label+ element_spec ; [comment]

case_label           : case const_expr : [comment]
                     | default : [comment]

element_spec         : type_spec declarator

enum_type            : enum identifier { identifier
                       {, identifier}* [comment] }

sequence_type        : sequence < simple_type_spec , const_expr >
                     | sequence < simple_type_spec >

string_type          : string < const_expr >
                     | string

attr_dcl             : [readonly] attribute simple_type_spec
                       declarator {, declarator}*
```

```
op_dcl              : [oneway] op_type_spec [stars] identifier
                       parameter_dcls [raises_expr] [context_expr]

op_type_spec        : simple_type_spec
                    | void

parameter_dcls      : ( param_dcl {, param_dcl}* [comment] )
                    | ( )

param_dcl           : param_attribute simple_type_spec declarator

param_attribute     : in
                    | out
                    | inout

raises_expr         : raises ( scope_name+ )

context_expr        : context ( context_string {, context_string}* )

implementation_body : implementation [comment]
                      { [comment] implementation+ }

implementation      : modifier_stm
                    | pragma_stm
                    | passthru
                    | member

pragma_stm          : #pragma modifier modifier_stm
                    | #pragma somtemittypes on
                    | #pragma somtemittypes off

modifier_stm        : smidentifier : [modifier {, modifier}*] ; [comment]
                    | modifier ; [comment]

modifier            : smidentifier
                    | smidentifier = modifier_value

modifier_value      : smidentifier
                    | string_literal
                    | integer_literal
                    | keyword

passthru            : passthru identifier = string_literal+ ; [comment]

smidentifier        : identifer
                    | _identifier

stars               | *+
```

Appendix C. Implementing Sockets Subclasses

Contents

Appendix C. Implementing Sockets Subclasses

Distributed SOM (DSOM) requires basic message services for inter-process communications. The Event Management Framework must be integrated with the same communication services in order to handle communications events.

To maximize their portability to a wide variety of local area network transport protocols, the DSOM and Event Management Frameworks have been written to use a *common communications interface*, which is implemented by one or more SOM class libraries using available local protocols.

The common communications interface is based on the “sockets” interface used with TCP/IP, since its interface and semantics are fairly widespread and well understood. The IDL interface is named **Sockets**. There is no implementation associated with the **Sockets** interface by default; specific protocol implementations are supplied by subclass implementations.

Note: The **Sockets** classes supplied with the SOMobjects Developer Toolkit and run-time packages are *only* intended to support the DSOM and Event Management Frameworks. These class implementations are <u>*not* intended for general application usage</u>.

The SOMobjects Workstation run-time package on AIX or OS/2 includes the **TCPIPSockets** class for TCP/IP. The SOMobjects Workgroup run-time package on AIX includes the **TCPIPSockets** class for TCP/IP, and the **IPXSockets** class for Netware IPX/SPX. On OS/2, the Workgroup run-time package includes the class **TCPIPSockets** for TCP/IP, the class **NBSockets** for NetBIOS, and the class **IPXSockets** for Netware IPX/SPX.

Application developers may need to develop their own **Sockets** subclass if the desired transport protocol or product version is not one of those supported by the SOMobjects run-time packages. This appendix explains how to approach the implementation of a **Sockets** subclass, if necessary. <u>Warning</u>: this may be a non-trivial exercise!

Sockets IDL interface

The base **Sockets** interface is expressed in IDL in the file **somssock.idl**, listed below. There is a one-to-one mapping between TCP/IP socket APIs and the methods defined in the **Sockets** interface.

Please note the following:

- The semantics of each of the **Sockets** methods must be that of the corresponding TCP/IP call. Currently, only Internet address family (AF_INET) addresses are used by the frameworks.

  (The TCP/IP sockets API is not documented as part of the SOMobjects Developer Toolkit. The implementor is referred to the programming references for IBM TCP/IP for AIX or OS/2, or to similar references that describe the sockets interface for TCP/IP.)

- Data types, constants, and macros which are part of the **Sockets** interface are defined in a C include file, **soms.h**. This file is supplied with the SOMobjects Toolkit, and is not shown in this manual.

- The **Sockets** interface is expressed in terms of a 32-bit implementation.

- Some of the method parameters and return values are expressed using pointer types, for example:

  ```
  hostent *somsGethostent ();
  ```

This has been done to map TCP/IP socket interfaces as directly as possible to their IDL equivalent. (Use of strict CORBA IDL was not a primary goal for the **Sockets** interface, since it is only used internally by the frameworks.)

- The **Sockets** class and its subclasses are *single instance* classes.

Following is a listing of the file **somssock.idl**. Each socket call is briefly described with a comment.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials – Property of IBM

#ifndef somssock_idl
#define somssock_idl

#include <somobj.idl>
#include <snglicls.idl>

interface Sockets : SOMObject
{
    //# The following typedefs are fully defined in <soms.h>.
    typedef SOMFOREIGN sockaddr;
    #pragma modifier sockaddr : impctx="C", struct;
    typedef SOMFOREIGN iovec;
    #pragma modifier iovec : impctx="C", struct;
    typedef SOMFOREIGN msghdr;
    #pragma modifier msghdr : impctx="C", struct;
    typedef SOMFOREIGN fd_set;
    #pragma modifier fd_set : impctx="C", struct;
    typedef SOMFOREIGN timeval;
    #pragma modifier timeval : impctx="C", struct;
    typedef SOMFOREIGN hostent;
    #pragma modifier hostent : impctx="C", struct;
    typedef SOMFOREIGN servent;
    #pragma modifier servent : impctx="C", struct;
    typedef SOMFOREIGN in_addr;
    #pragma modifier in_addr : impctx="C", struct;

    long somsAccept (in long s, out sockaddr name, out long namelen);
    // Accept a connection request from a client.

    long somsBind (in long s, inout sockaddr name, in long namelen);
    // Binds a unique local name to the socket with descriptor s.

    long somsConnect (in long s, inout sockaddr name,
                      in long namelen);
    // For streams sockets, attempts to establish a connection
    // between two sockets.  For datagram sockets, specifies the
    // socket's peer.

    hostent *somsGethostbyaddr (in char *addr, in long addrlen,
                                in long domain);
    // Returns a hostent structure for the host address specified on
    // the call.

    hostent *somsGethostbyname (in string name);
    // Returns a hostent structure for the host name specified on
    // the call.

    hostent *somsGethostent ();
    // Returns a pointer to the next entry in the hosts file.
```

```
unsigned long somsGethostid ();
// Returns the unique identifier for the current host.

long somsGethostname (in string name, in long namelength);
// Retrieves the standard host name of the local host.

long somsGetpeername (in long s, out sockaddr name,
                      out long namelen);
// Gets the name of the peer connected to socket s.

servent *somsGetservbyname (in string name, in string protocol);
// Retrieves an entry from the /etc/services file using the
// service name as a search key.

long somsGetsockname (in long s, out sockaddr name,
                      out long namelen);
// Stores the current name for the socket specified by the s
// parameter into the structure pointed to by the name
// parameter.

long somsGetsockopt (in long s, in long level, in long optname,
                     in char *optval, out long option);
// Returns the values of socket options at various protocol
// levels.

unsigned long somsHtonl (in unsigned long a);
// Translates an unsigned long integer from host-byte order to
// network-byte order.

unsigned short somsHtons (in unsigned short a);
// Translates an unsigned short integer from host-byte order to
// network-byte order.

long somsIoctl (in long s, in long cmd, in char *data,
                in long length);
// Controls the operating characteristics of sockets.

unsigned long somsInet_addr (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as internet addresses.

unsigned long somsInet_lnaof (in in_addr addr);
// Breaks apart the internet address and returns the local
// network address portion.

in_addr somsInet_makeaddr (in unsigned long net,
                           in unsigned long lna);
// Takes a network number and a local network address and
// constructs an internet address.

unsigned long somsInet_netof (in in_addr addr);
// Returns the network number portion of the given internet
// address.

unsigned long somsInet_network (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as network numbers.

string somsInet_ntoa (in in_addr addr);
// Returns a pointer to a string expressed in the dotted-decimal
// notation.
```

```
long somsListen (in long s, in long backlog);
// Creates a connection request queue of length backlog to queue
// incoming connection requests, and then waits for incoming
// connection requests.

unsigned long somsNtohl (in unsigned long a);
// Translates an unsigned long integer from network-byte order
// to host-byte order.

unsigned short somsNtohs (in unsigned short a);
// Translates an unsigned short integer from network-byte order
// to host-byte order.

long somsReadv (in long s, inout iovec iov, in long iovcnt);
// Reads data on socket s and stores it in a set of buffers
// described by iov.

long somsRecv (in long s, in char *buf, in long len,
               in long flags);
// Receives data on streams socket s and stores it in buf.

long somsRecvfrom (in long s, in char *buf, in long len,
    in long flags, out sockaddr name, out long namelen);
// Receives data on datagram socket s and stores it in buf.

long somsRecvmsg (in long s, inout msghdr msg, in long flags);
// Receives messages on a socket with descriptor s and stores
// them in an array of message headers.

long somsSelect (in long nfds, inout fd_set readfds,
                 inout fd_set writefds, inout fd_set exceptfds,
                 inout timeval timeout);
// Monitors activity on a set of different sockets until a
// timeout expires, to see if any sockets are ready for reading
// or writing, or if an exceptional condition is pending.

long somsSend (in long s, in char *msg, in long len,
               in long flags);
// Sends msg on streams socket s.

long somsSendmsg (in long s, inout msghdr msg, in long flags);
// Sends messages passed in an array of message headers on a
// socket with descriptor s.

long somsSendto (in long s, inout char msg, in long len,
               in long flags, inout sockaddr to, in long tolen);
// Sends msg on datagram socket s.

long somsSetsockopt (in long s, in long level, in long optname,
                     in char *optval, in long optlen);
// Sets options associated with a socket.

long somsShutdown (in long s, in long how);
// Shuts down all or part of a full-duplex connection.

long somsSocket (in long domain, in long type,
                 in long protocol);
// Creates an endpoint for communication and returns a socket
// descriptor representing the endpoint.

long somsSoclose (in long s);
// Shuts down socket s and frees resources allocated to the
// socket.
```

```
    long somsWritev (in long s, inout iovec iov, in long iovcnt);
    // Writes data on socket s.  The data is gathered from the
    // buffers described by iov.

    attribute long serrno;
    // Used to pass error numbers.

#ifdef __SOMIDL__
    implementation
    {
    releaseorder:
        somsAccept, somsBind, somsConnect, somsGethostbyaddr,
        somsGethostbyname, somsGethostent, somsGethostid,
        somsGethostname, somsGetpeername, somsGetsockname,
        somsGetsockopt, somsHtonl, somsHtons, somsIoctl,
        somsInet_addr, somsInet_lnaof, somsInet_makeaddr,
        somsInet_netof, somsInet_network, somsInet_ntoa,
        somsListen, somsNtohl, somsNtohs, somsReadv,
        somsRecv, somsRecvfrom, somsRecvmsg, somsSelect,
        somsSend, somsSendmsg, somsSendto, somsSetsockopt,
        somsShutdown, somsSocket, somsSoclose, somsWritev,
        _set_serrno, _get_serrno, somsGetservbyname;

    //# Class modifiers
    callstyle=idl;
    metaclass = SOMMSingleInstance;
    majorversion=1; minorversion=1;
    dll="soms.dll";
    };
#endif /* __SOMIDL__ */
};
#endif  /* somssock_idl */
```

IDL for a Sockets subclass

Sockets subclasses inherit their entire interface from **Sockets.** All methods are overridden.

For example, here is a listing of the **TCPIPSockets** IDL description.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials - Property of IBM

#ifndef tcpsock_idl
#define tcpsock_idl

#include <somssock.idl>
#include <snglicls.idl>

interface TCPIPSockets : Sockets
{
#ifdef __SOMIDL__
    implementation
    {
    //# Class modifiers
    callstyle=idl;
    majorversion=1; minorversion=1;
    dllname="somst.dll";
    metaclass=SOMMSingleInstance;
```

```
    //# Method modifiers
    somsAccept: override;
    somsBind: override;
    somsConnect: override;
    somsGethostbyaddr: override;
    somsGethostbyname: override;
    somsGethostent: override;
    somsGethostid: override;
    somsGethostname: override;
    somsGetpeername: override;
    somsGetservbyname: override;
    somsGetsockname: override;
    somsGetsockopt: override;
    somsHtonl: override;
    somsHtons: override;
    somsIoctl: override;
    somsInet_addr: override;
    somsInet_lnaof: override;
    somsInet_makeaddr: override;
    somsInet_netof: override;
    somsInet_network: override;
    somsInet_ntoa: override;
    somsListen: override;
    somsNtohl: override;
    somsNtohs: override;
    somsReadv: override;
    somsRecv: override;
    somsRecvfrom: override;
    somsRecvmsg: override;
    somsSelect: override;
    somsSend: override;
    somsSendmsg: override;
    somsSendto: override;
    somsSetsockopt: override;
    somsShutdown: override;
    somsSocket: override;
    somsSoclose: override;
    somsWritev: override;
    _set_serrno: override;
    _get_serrno: override;
    };
#endif /* __SOMIDL__ */
};

#endif  /* tcpsock_idl */
```

Implementation considerations

- Only the AF_INET address family must be supported. That is, the DSOM and Event Manager frameworks both use Internet addresses and port numbers to refer to specific sockets.
- On OS/2, the SOMobjects run-time libraries were built using the C Set/2 32-bit compiler. If the underlying subclass implementation uses a 16-bit subroutine library, conversion of the method call arguments may be required. (This mapping of arguments is often referred to as “thunking.”)

- **Sockets** subclasses to be used in multi-threaded environments should be made thread-safe. That is, it is possible that concurrent threads may make calls on the (single) **Sockets** object, so data structures must be protected within critical regions, as appropriate.
- Valid values for the **serrno** attribute are defined in the file **soms.h**. The subclass implementation should map local error numbers into the appropriate corresponding **Sockets** error numbers.

Example code

The following code fragment shows an example of the implementation of the **somsBind** method of the **TCPIPSockets** subclass, for both AIX and OS/2. The sample illustrates that, for TCP/IP, the implementation is basically a one-to-one mapping of **Sockets** methods onto TCP/IP calls. For other transport protocols, the mapping from the socket abstraction to the protocol's API may be more difficult.

For AIX, the mapping from **Sockets** method to TCP/IP call is trivial.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
     long rc;

     TCPIPSocketsMethodDebug("TCPIPSockets","somsBind");

     rc = (long) bind((int)s, name, (int)namelen);

     if (rc == -1)
         __set_serrno(somSelf, ev, errno);

     return rc;
}
```

On OS/2, however, the TCP/IP Release 1.2.1 library is a 16-bit library. Consequently, many of the method calls require conversion ("thunking") of 32-bit parameters into 16-bit parameters, before the actual TCP/IP calls can be invoked. For example, the function prototype for the **somsBind** method is defined as:

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen);
```

whereas the file **socket.h** on OS/2 declares the **bind** function with the following prototype:

```
short _Far16 _Cdecl bind(short /*s*/, void * _Seg16 /*name*/,
                         short /*len*/);
```

In this case, the pointer to the "name" structure, passed as a 32-bit address, cannot be used directly in the **bind** call: a 16-bit address must be passed instead. This can be accomplished by dereferencing the 32-bit pointer provided by the "name" parameter in the **somsBind** call, copying the caller's Sockets_sockaddr structure into a local structure ("name16"), and then passing the address of the local structure ("&name16") as a 16-bit address in the **bind** call.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
    long rc;
    Sockets_sockaddr name16;

    TCPIPSocketsMethodDebug("TCPIPSockets","somsBind");

    /* copy user's parameter into a local structure */
    memcpy ((char *)&name16, (char *)((sockaddr32 *)name), namelen);
    rc = (long) bind((short)s, (void *)&name16, (short)namelen);

    if (rc == -1)
        __set_serrno(somSelf, ev, tcperrno());

    return rc;
}
```

Glossary

Note: In the following definitions, words shown in *italics* are terms for which separate glossary entries are also defined.

abstract class
A *class* that is not designed to be instantiated, but serves as a *base class* for the definition of subclasses. Regardless of whether an abstract class inherits *instance data* and *methods* from *parent classes*, it will always introduce methods that must be *overridden* in a *subclass,* in order to produce a class whose objects are semantically valid.

affinity group An array of *class objects* that were all registered with the *SOMClassMgr* object during the dynamic loading of a *class*. Any class is a member of at most one affinity group.

ancestor class
A *class* from which another class inherits *instance methods, attributes,* and *instance variables*, either directly or indirectly. A direct descendant of an ancestor class is called a *child class, derived class,* or *subclass.* A direct ancestor of a class is called a *parent class, base class*, or *superclass*.

aggregate type
A user-defined data type that combines basic types (such as, char, short, float, and so on) into a more complex type (such as structs, arrays, strings, sequences, unions, or enums).

apply stub A *procedure* corresponding to a particular *method* that accepts as arguments: the *object* on which the method is to be invoked, a pointer to a location in memory where the method's result should be stored, a pointer to the method's procedure, and the method's arguments in the form of a va_list. The apply stub extracts the arguments from the va_list, invokes the method with its arguments, and stores its result in the specified location. Apply stubs are registered with class objects when instance methods are defined, and are invoked using the somApply function. Typically, *implementations* that *override* somDispatch call somApply to invoke a method on a va_list of arguments.

attribute A specialized syntax for declaring "set" and "get" methods. Method names corresponding to attributes always begin with "_set_" or "_get_". An attribute name is declared in the body of the *interface statement* for a class. Method procedures for get/set methods are automatically defined by the *SOM Compiler* unless an attribute is declared as "noget/noset". Likewise, a corresponding *instance variable* is automatically defined unless an attribute is declared as "nodata". IDL also supports "readonly" attributes, which specify only a "get" method. (Contrast an attribute with an *instance variable.*)

auxiliary class data structure
A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is <*className*>CClassData. Its first component (parentMtab) is a list of *parent-class method tables* (used to support efficient parent method calls). Its second component (instanceDataToken) is the *instance token* for the class (generally used to locate the *instance data* introduced by *method procedures* that implement *methods* defined by the class).

base class See *parent class.*

behavior (of an object)
The *methods* that an *object* responds to. These methods are those either introduced or inherited by the *class* of the object. See also *state.*

bindings Language-specific macros and procedures that make implementing and using SOM classes more convenient. These bindings offer a convenient interface to SOM that is tailored to a particular programming language. The *SOM Compiler* generates binding files for C and C++. These binding files include an *implementation template* for the class and two header files, one to be included in the class's implementation file and the other in client programs.

BOA (basic object adapter) class
A CORBA *interface* (represented as an *abstract class* in DSOM), which defines generic *object-adapter (OA)* methods that a *server* can use to register itself and its *objects* with an *ORB (object request broker)*. See also *SOMOA (SOM object adapter) class.*

callback A user-provided procedure or method to the Event Management Framework that gets invoked when a registered event occurs. (See also *event*).

casted dispatching
A form of method dispatching that uses *casted method resolution*; that is, it uses a designated ancestor class of the actual *target object's* class to determine what procedure to call to execute a specified method.

casted method resolution
A *method resolution* technique that uses a *method procedure* from the *method table* of an ancestor of the *class* of an *object* (rather than using a procedure from the method table of the object's own class).

child class A class that inherits *instance methods, attributes,* and *instance variables* directly from another class, called the *parent class, base class*, or *superclass,* or indirectly from an *ancestor class.* A child class may also be called a *derived class* or *subclass.*

class A way of categorizing *objects* based on their behavior (the *methods* they support) and shape (memory layout). A class is a definition of a generic object. In SOM, a class is also a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior. The specification of what comprises the shape and behavior of a set of objects is referred to as the "definition" of a class. New classes are defined in terms of existing classes through a technique known as *inheritance*. See also *class object.*

class variable *Instance data* of a *class object*. All instance data of an *object* is defined (through either introduction or *inheritance*) by the object's class. Thus, class variables are defined by *metaclasses*.

class data structure
A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is <*className*>ClassData. Its first component (classObject) is a pointer to the corresponding *class object*. The remaining components (named after the *instance methods* and *instance variables*) are *method tokens* or *data tokens*, in order as specified by the class's implementation. Data tokens are only used to support data (public and private) introduced by classes declared using *OIDL*; IDL *attributes* are supported with method tokens.

class manager
An *object* that acts as a run-time registry for all SOM *class objects* that exist within the current process and which assists in the dynamic loading and unloading of class libraries. A class implementor can define a customized class manager by subclassing *SOMClassMgr* class to replace the SOM-supplied SOMClassMgrObject. This is done to augment the functionality of the default class-management registry (for example, to coordinate the automatic quiescing and unloading of classes).

class method (Also known as *factory method* or *constructor*.) A class method is a *method* that a *class object* responds to (as opposed to an *instance method*). A class method that class <X> responds to is provided by the *metaclass* of class <X>. Class methods are executed without requiring any *instances* of class <X> to exist, and are frequently used to create instances of the class.

class object The run-time *object* representing a SOM *class.* In SOM, a class object can perform the same behavior common to all *objects*, inherited from *SOMObject*.

client code (Or *client program* or *client*.) An application program, written in the programmer's preferred language, which invokes *methods* on *objects* that are *instances* of SOM *classes*. In DSOM, this could be a program that invokes a method on a remote object.

constructor See *class method*.

context expression
An optional expression in a method's IDL declaration, specifying identifiers whose value (if any) can be used during SOM's *method resolution* process and/or by the *target object* as it executes the *method procedure*. If a context expression is specified, then a related Context parameter is required when the method is invoked. (This Context parameter is an *implicit parameter* in the IDL specification of the method, but it is an explicit parameter of the method's procedure.) No SOM-supplied methods require context parameters.

CORBA The Common Object Request Broker Architecture established by the Object Management Group. IBM's *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

data token A value that identifies a specific *instance variable* within an *object* whose *class inherits* the instance variable (as a result of being derived, directly or indirectly, from the class that introduces the instance variable). An object and a data token are passed to the SOM run-time procedure, somDataResolve, which returns is a pointer to the specific instance variable corresponding to the data token. (See also *instance token.*)

derived class See *subclass* and *subclassing.*

derived metaclass
(Or *SOM-derived metaclass*.) A *metaclass* that SOM creates automatically (often even when the *class* implementor specifies an explicit metaclass) as needed to ensure that, for any code that executes without *method-resolution* error on an *instance* of a given class, the code will similarly execute without method-resolution error on instances of any *subclass* of the given class. SOM's ability to derive such metaclasses is a fundamental necessity in order to ensure binary compatibility for client programs despite any subsequent changes in class *implementations*.

descriptor (Or *method descriptor*.) An ID representing the identifier of a *method* definition or an *attribute* definition in the Interface Repository. The IR definition contains information about the method's return type and the type of its arguments.

directive A message (a pre-defined character constant) received by a *replica* from the Replication Framework. Indicates a potential failure situation.

dirty object A persistent *object* that has been modified since it was last written to persistent storage.

dispatch-function resolution
Dispatch-function resolution is the slowest, but most flexible, of the three *method-resolution* techniques SOM offers. Dispatch functions permit method resolution to be based on arbitrary rules associated with an *object*'s *class*. Thus, a class implementor has complete freedom in determining how methods invoked on its *instances* are resolved. See also *dispatch method* and *dynamic dispatching.*

dispatch method
A *method* (such as somDispatch or somClassDispatch) that is invoked (and passed an argument list and the ID of another method) in order to determine the appropriate *method procedure* to execute. The use of dispatch methods facilitates *dispatch-function resolution* in SOM applications and enables method invocation on remote objects in DSOM applications. See also *dynamic dispatching*.

dynamic dispatching
Method dispatching using *dispatch-function resolution*; the use of dynamic *method resolution* at run time. See also *dispatch-function resolution* and *dynamic method.*

Dynamic Invocation Interface (DII)
The *CORBA*-specified *interface*, implemented in DSOM, that is used to dynamically build requests on remote *objects*. Note that DSOM applications can also use the somDispatch method for *dynamic method* calls when the object is remote. See also *dispatch method.*

dynamic method
A method that is not declared in the IDL *interface statement* for a *class* of *objects*, but is added to the *interface* at run time, after which *instances* of the class (or of its *subclasses*) will respond to the registered dynamic method. Because dynamic methods are not declared, *usage bindings* for SOM classes cannot support their use; thus, *offset method resolution* is not available. Instead, *name-lookup* or *dispatch-function method resolution* must be used to invoke dynamic methods. (There are currently no known uses of dynamic methods by any SOM applications.) See also *method* and *static method.*

encapsulation
An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are only required to know the *interface* of a *class* (the signatures of its *methods* and the names of its *attributes*) in order to use the class's methods and attributes.

encoder/decoder
In the Persistence Framework, a *class* that knows how to read/write the persistent object format of a *persistent object*. Every persistent object is associated with an Encoder/Decoder, and an encoder/decoder object is created for each *attribute* and *instance variable*. An Encoder/Decoder is supplied by the Persistence Framework by default, or an application can define its own.

entry class In the Emitter Framework, a *class* that represents some syntactic unit of an *interface* definition in the *IDL source file.*

Environment parameter

A *CORBA*-required parameter in all *method procedures*, it represents a memory location where exception information can be returned by the *object* of a method invocation. [Certain methods are exempt (when the class contains a modifier of callstyle=oidl), to maintain upward compatibility for client programs written using an earlier release.]

emitter

Generically, a program that takes the output from one system and converts the information into a different form. Using the Emitter Framework, selected output from the *SOM Compiler* (describing each syntactic unit in an *IDL source file*) is transformed and formatted according to a user-defined template. Example emitter output, besides the implementation template and language bindings, might include reference documentation, class browser descriptions, or “pretty” printouts.

event

The occurrence of a condition, or the beginning or ending of an activity that is of interest to an application. Examples are elapse of a time interval, sending or receiving of a message, and opening or closing a file. (See also *event manager* and *callback*.)

event manager (EMan)

The chief component of the Event Management Framework that registers interest in various *events* from calling modules and informs them through *callbacks* when those events occur.

factory method

See *class method*.

ID

See *somId.*

IDL source file

A user-written .idl file, expressed using the syntax of the *Interface Definition Language* (IDL), which describes the *interface* for a particular *class* (or classes, for a *module*). The IDL source file is processed by the *SOM Compiler* to generate the *binding files* specific to the programming languages of the class implementor and the client application. (This file may also be called the “IDL file,” the “source file,” or the “interface definition file.”)

implementation

(Or *object implementation*.) The specification of what *instance variables* implement an *object*'s *state* and what *procedures* implement its *methods* (or *behaviors*). In DSOM, a remote object's implementation is also characterized by its *server implementation* (a program).

Implementation Repository

A database used by DSOM to store the implementation definitions of DSOM *servers*.

implementation statement

An optional declaration within the body of the *interface* definition of a *class* in a SOM *IDL source file,* specifying information about how the class will be implemented (such as, version numbers for the class, overriding of inherited methods, or type of method resolution to be supported by particular methods). This statement is a SOM-unique statement; thus, it must be preceded by the term “#ifdef __SOMIDL__” and followed by “#endif”. See also *interface declaration.*

implementation template

A template file containing *stub procedures* for *methods* that a *class* introduces or *overrides*. The implementation template is one of the *binding files* generated by the *SOM Compiler* when it processes the *IDL source file* containing class *interface declarations.* The class implementor then customizes the *implementation,* by adding language-specific code to the *method procedures*.

implicit method parameter

A *method* parameter that is not included in the IDL *interface* specification of a method, but which is a parameter of the *method's* *procedure* and which is required when the method is invoked from a *client program*. Implicit parameters include the required *Environment* parameter indicating where exception information can be returned, as well as a *Context* parameter, if needed.

incremental update

A revision to an *implementation template* file that results from reprocessing of the *IDL source file* by the *SOM Compiler*. The updated implementation file will contain new *stub procedures*, added comments, and revised *method prototypes* reflecting changes made to the *method* definitions in the IDL specification. Importantly, these updates do not disturb existing code that the class implementor has defined for the prior method procedures.

inheritance

The technique of defining one *class* (called a *subclass, derived class,* or *child class*) as incremental differences from another class (called the *parent class, base class, superclass,* or *ancestor class*). From its parents, the subclass inherits variables and *methods* for its *instances*. The subclass can also provide additional *instance variables* and methods. Furthermore, the subclass can provide new procedures for implementing inherited methods. The subclass is then said to *override* the parent class's methods. An overriding method procedure can elect to call the parent class's *method procedure*. (Such a call is known as a *parent method call*.)

inheritance hierarchy

The sequential relationship from a root class to a subclass, through which the subclass inherits *instance methods, attributes,* and *instance variables* from all of its ancestors, either directly or indirectly. The root class of all SOM classes is SOMObject.

instance

(Or *object instance* or just *object*.) A specific object, as distinguished from a *class* of objects. See also *object.*

instance method

A method valid for an object *instance* (as opposed to a *class method*, which is valid for a *class object*). An instance method that an object responds to is defined by its class or inherited from an ancestor class.

instance token

A *data token* that identifies the first *instance variable* among those introduced by a given *class*. The somGetInstanceToken *method* invoked on a *class object* returns that class's instance token.

instance variables

(Or, *instance data*.) Variables declared for use within the *method procedures* of a *class*. An instance variable is declared within the body of the *implementation statement* in a SOM *IDL source file*. An instance variable is "private" to the class and should not be accessed by a client program. (Contrast an instance variable with an *attribute.*)

interface

The information that a *client* must know to use a *class* — namely, the names of its *attributes* and the signatures of its *methods*. The interface is described in a formal language (the *Interface Definition Language*, IDL) that is independent of the programming language used to implement the class's methods.

interface declaration
(Or *interface statement.*) The statement in the *IDL source file* that specifies the name of a new class and the names of its *parent class*(es). The “body” of the interface declaration defines the *signature* of each new *method* and any *attribute*(s) associated with the class. In SOM IDL, the body may also include an *implementation statement* (where *instance variables* are declared or a *modifier* is specified, for example to *override* a method).

Interface Definition Language (IDL)
The formal language (independent of any programming language) by which the *interface* for a *class* of *objects* is defined in a .idl file, which the *SOM Compiler* then interprets to create an *implementation template* file and *binding* files. SOM’s Interface Definition Language is fully compliant with standards established by the Object Management Group’s Common Object Request Broker Architecture (*CORBA*).

Interface Repository (IR)
The database that SOM optionally creates to provide persistent storage of objects representing the major elements of *interface* definitions. Creation and maintenance of the IR is based on information supplied in the *IDL source file.* The SOM IR Framework supports all interfaces described in the *CORBA* standard.

Interface Repository Framework
A set of *classes* that provide *methods* whereby executing programs can access the persistent objects of the *Interface Repository* to discover everything known about the programming *interfaces* of SOM classes.

macro An alias for executing a sequence of hidden instructions; in SOM, typically the means of executing a command known within a *binding file* created by the *SOM Compiler*.

metaclass A *class* whose *instances* are classes. In SOM, any class descended from *SOMClass* is a metaclass. The *methods* a class inherits from its metaclass are sometimes called *class methods* (in Smalltalk) or *factory methods* (in Objective-C) or *constructors*. See also *class method.*

metaclass incompatibility
A situation where a *subclass* does not include all of the *class variables* or respond to all of the *class methods* of its *ancestor classes*. This situation can easily arise in *OOP* systems that allow programmers to explicitly specify *metaclasses*, but is not allowed to occur in SOM. Instead, SOM automatically prevents this by creating and using *derived metaclasses* whenever necessary.

method A combination of a *procedure* and a name, such that many different procedures can be associated with the same name. In object-oriented programming, invoking a method on an *object* causes the object to execute a specific *method procedure*. The process of determining which method procedure to execute when a method is invoked on an object is called *method resolution*. (The *CORBA* standard uses the term “operation” for method invocation). SOM supports two different kinds of methods: static methods and dynamic methods. See also *static method* and *dynamic method.*

method descriptor See *descriptor*.

method ID A number representing a zero-terminated string by which SOM uniquely represents a *method* name. See also *somId.*

method procedure
A function or procedure, written in an arbitrary programming language, that implements a *method* of a *class*. A method procedure is defined by the class implementor within the *implementation template* file generated by the *SOM Compiler.*

method prototype
A *method* declaration that includes the types of the arguments. Based on method definitions in an *IDL source file*, the *SOM Compiler* generates method prototypes in the *implementation template*. A class implementor uses the method prototype as a basis for writing the corresponding *method procedure* code. The method prototype also shows all arguments and their types that are required to invoke the method from a *client program*.

method resolution
The process of selecting a particular *method procedure*, given a *method* name and an object *instance*. The process results in selecting the particular function/procedure that implements the abstract method in a way appropriate for the designated object. SOM supports a variety of method-resolution mechanisms, including *offset method resolution, name-lookup resolution,* and *dispatch-function resolution*.

method table A table of pointers to the *method procedures* that implement the *methods* that an *object* supports. See also *method token.*

method token A value that identifies a specific *method* introduced by a *class*. A method token is used during *method resolution* to locate the *method procedure* that implements the identified method. The two basic method-resolution procedures are somResolve (which takes as arguments an *object* and a method token, and returns a pointer to a procedure that implements the identified method on the given object) and somClassResolve (which takes as arguments a *class* and a method token, and returns a pointer to a procedure that implements the identified method on an instance of the given class).

modifier Any of a set of statements that control how a *class*, an *attribute*, or a *method* will be implemented. Modifiers can be defined in the *implementation statement* of a SOM *IDL source file*. The implementation statement is a SOM-unique extension of the *CORBA* specification. [User-defined modifiers can also be specified for use by user-written emitters or to store information in the *Interface Repository*, which can then be accessed via methods provided by the *Interface Repository Framework*.]

module The organizational structure required within an *IDL source file* that contains *interface declarations* for two (or more) classes that are not a class–metaclass pair. Such *interfaces* must be grouped within a module declaration.

multiple inheritance
The situation in which a *class* is derived from (and inherits *interface* and *implementation* from) multiple parent classes.

name-lookup method resolution
Similar to the *method resolution* techniques employed by Objective-C and Smalltalk. It is slower than *offset resolution* (roughly two to three times the cost of an ordinary procedure call). Name-lookup resolution, unlike offset resolution, can be used when the name of the method to be invoked is not known until run time, or the method is added to the class interface at run time, or the name of the class introducing the method is not known until run time.

naming scope See *scope*.

object (Or *object instance* or just *instance*.) An entity that has *state* (its data values) and *behavior* (its *methods*). An object is one of the elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way to encapsulate state and behavior. *Encapsulation* permits many aspects of the *implementation* of an object to change without affecting client programs that depend on the object's behavior. In SOM, objects are created by other objects called *classes*.

object adapter (OA)
A *CORBA* term denoting the primary interface a *server implementation* uses to access *ORB* functions; in particular, it defines the mechanisms that a server uses to interact with DSOM, and vice versa. This includes server activation/deactivation, dispatching of *methods*, and authentication of the *principal* making a call. The basic object adapter described by CORBA is defined by the *BOA (basic object adapter) abstract class*; DSOM's primary object adapter implementation is provided by the *SOMOA* (*SOM Object Adapter*) *class*.

object definition See *class*.

object implementation See *implementation.*

object instance See *instance* and *object*.

object reference
A *CORBA* term denoting the information needed to reliably identify a particular *object*. This concept is implemented in DSOM with a *proxy object* in a *client* process, or a *SOMDObject* in a *server* process. See also *proxy object* and *SOMDObject.*

object request broker (ORB) See *ORB*.

offset method resolution
The default mechanism for performing *method resolution* in SOM, because it is the fastest (nearly as fast as an ordinary procedure call). It is roughly equivalent to the C++ "virtual function" concept. Using offset method resolution requires that the name of the *method* to be invoked must be known at compile time, the name of the *class* that introduces the method must be known at compile time (although not necessarily by the programmer), and the method to be invoked must be a *static method*.

OIDL The original language used for declaring SOM *classes*. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM release 2, but it does not include the ability to specify *multiple inheritance* classes.

one-copy serializable
The consistency property of the Replication Framework which states that the concurrent execution of *methods* on a *replicated object* is equivalent to the serial execution of those same methods on a nonreplicated object.

OOP An acronym for "object-oriented programming."

operation See *method.*

operation logging
In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the execution of a *method* that updates the *object* is repeated at the site of each replica.

ORB (object request broker)

A *CORBA* term designating the means by which *objects* transparently make requests (that is, invoke *methods*) and receive responses from objects, whether they are local or remote. With SOMobjects Developer Toolkit and Runtimes, this functionality is implemented in the DSOM Framework. Thus, the DSOM (Distributed SOM) system is an ORB. See also *BOA (basic object adapter) class* and *SOMOA (SOM object adapter) class.*

override (Or *overriding method.*) The technique by which a *class* replaces (redefines) the *implementation* of a *method* that it inherits from one of its *parent classes*. An overriding method can elect to call the parent class's *method procedure* as part of its own implementation. (Such a call is known as a *parent method call.*)

parent class A *class* from which another class inherits *instance methods*, *attributes*, and *instance variables*. A parent class is sometimes called a *base class* or *superclass*.

parent method call

A technique where an *overriding method* calls the *method procedure* of its *parent class* as part of its own *implementation*.

persistent object

An *object* whose *state* can be preserved beyond the termination of the *process* that created it. Typically, such objects are stored in files.

polymorphism

An object-oriented programming feature that may take on different meanings in different systems. Under various definitions of polymorphism, (a) a *method* or *procedure* call can be executed using arguments of a variety of types, or (b) the same variable can assume values of different types at different times, or (c) a method name can denote more than one *method procedure.* The SOM system reflects the third definition (for example, when a SOM class *overrides* a *parent class* definition of a method to change its behavior). The term literally means "having many forms."

principal The user on whose behalf a particular (remote) *method* call is being performed.

procedure A small section of code that executes a limited, well-understood task when called from another program. In SOM, a *method procedure* is often referred to as a procedure. See also *method procedure.*

process A series of instructions (a program or part of a program) that a computer executes in a multitasking environment.

proxy object In DSOM, a SOM *object* in the *client*'s address space that represents a remote *object.* The proxy object has the same *interface* as the remote object, but each *method* invoked on the proxy is *overridden* by a *dispatch method* that forwards the invocation request to the remote object. Under DSOM, a proxy object is created dynamically and automatically in the client whenever a remote method returns a pointer to an object that happens to be remote.

readers and writers

In the Replication Framework, different processes can access the same replicated object in different modes. A "reader" is a process that does not intend to update the object, but wants to continually watch the object as other processes update it. A "writer" is a process that wants to update the object, as well as continually watch the updates performed by others.

receiver See *target object.*

redispatch stub
A *procedure*, corresponding to a particular *method*, which has the same *signature* as the method's procedure but which invokes somDispatch to dispatch the method. The somOverrideMtab method can be used to replace the procedure pointers in a *class*'s *method table* with the corresponding redispatch stubs. This is done when *overriding* somDispatch to customize *method resolution* so that all *static method* invocations will be routed through somDispatch for selection of an appropriate *method procedure*. (*Dynamic methods* have no entries in the method table, so they cannot be supported with redispatch functionality.)

reference data
Application-specific data that a *server* uses to identify or describe an *object* in DSOM. The data, represented by a sequence of up to 1024 bytes, is registered with DSOM when a server creates an *object reference*. A server can later ask DSOM to return the reference data associated with an object reference. See also *object reference.*

replica
When an object is replicated among a set of processes (using the Replication Framework), each process is said to have a replica of the object. From the view point of any application model, the replicas together represent a single object.

replicated object
An *object* for which *replicas* (copies) exist. See *replica.*

run-time environment
The data structures, objects, and global variables that are created, maintained, and used by the functions, procedures, and methods in the SOM run-time library.

scope
(Or *naming scope.*) That portion of a program within which an identifier name has "visibility" and denotes a unique variable. In SOM, an *IDL source file* forms a scope. An identifier can only be defined once within a scope; identifiers can be redefined within a nested scope. In a .idl file, modules, interface statements, structures, unions, methods, and exceptions form nested scopes.

serializable
See *one-copy serializable*.

server
(Or *server implementation.*) In DSOM, a *process*, running in a distributed environment, that executes the *implementation* of an *object*. DSOM provides a default server implementation that can dynamically load SOM *class* libraries, create SOM objects, and make those objects accessible to *clients*. Developers can also write application-specific servers for use with DSOM.

server object
In DSOM, every *server* has an *object* that defines *methods* for managing objects in that server. These methods include object creation, object destruction, and maintaining mappings between *object references* and the objects they reference. A server object must be an *instance* of the *class* SOMDServer (or one of its *subclasses*). See also *object reference* and *SOMDObject.*

shadowing
In the Emitter Framework, a technique that is required when any of the *entry classes* are subclassed. Shadowing causes instances of the new subclass(es) (rather than instances of the original entry classes) to be used as input for building the object graph, without requiring a recompile of emitter framework code. Shadowing is accomplished by using the macro SOM_SubstituteClass.

signature
The collection of types associated with a *method* (the type of its return value, if any, as well as the number, order, and type of each of its arguments).

sister class object
A duplicate of a *class object* that is created in order to save a copy of the class's original *method table* before replacing the method table to customize *method resolution.* The sister class object is created so that some original *method procedures* can be called by the replacement method procedures.

Sockets class A class that provides a common communications interface to Distributed SOM, the Replication Framework, and the Event Management Framework. The Sockets class provides the base interfaces (patterned after TCP/IP sockets); the *subclasses* TCPIPSockets, NBSockets, and IPXSockets provide actual implementations for TCP/IP, Netbios, and Netware IPX/SPX, respectively.

SOM Compiler
A tool provided by the SOM Toolkit that takes as input the interface definition file for a class (the .idl file) and produces a set of *binding files* that make it more convenient to implement and use SOM classes.

SOMClass One of the three primitive *class objects* of the SOM run-time environment. SOMClass is the root (meta)class from which all subsequent *metaclasses* are derived. SOMClass defines the essential *behavior* common to all SOM *class objects.*

SOMClassMgr
One of the three primitive *class objects* of the SOM run-time environment. During SOM initialization, a single *instance* (*object*) of SOMClassMgr is created, called SOMClassMgrObject. This object maintains a directory of all SOM classes that exist within the current process, and it assists with dynamic loading and unloading of class libraries.

SOM-derived metaclass See *derived metaclass.*

SOMDObject The *class* that implements the notion of a *CORBA "object reference*" in DSOM. An *instance* of SOMDObject contains information about an object's *server implementation* and *interface,* as well as a user-supplied identifier.

somId A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somId's are used to represent *method* names, *class* names, and so forth.

SOMObject One of the three primitive *class objects* of the SOM run-time environment. SOMObject is the root class for all SOM (sub)classes. SOMObject defines the essential *behavior* common to all SOM *objects.*

SOMOA (SOM object adapter) class
In DSOM, a *class* that dispatches *methods* on a *server*'s *objects*, using the *SOM Compiler* and run-time support. The SOMOA class implements methods defined in the *abstract* BOA class (its *base class*). See also *BOA class.*

somSelf Within *method procedures* in the *implementation* file for a class, a parameter pointing to the *target object* that is an *instance* of the *class* being implemented. It is local to the *method procedure.*

somThis Within *method procedures,* a local variable that points to a data structure containing the *instance variables* introduced by the *class*. If no instance variables are specified in the SOM *IDL source file*, then the somThis assignment statement is commented out by the *SOM Compiler*.

state (of an object)
The data (*attributes, instance variables* and their values) associated with an *object.* See also *behavior.*

static method Any *method* that can be accessed through *offset method resolution*. Any method declared in the IDL specification of a class is a static method. See also *method* and *dynamic method.*

stub procedures
Method procedures in the *implementation template* generated by the *SOM Compiler*. They are procedures whose bodies are largely vacuous, to be filled in by the implementor.

subclass A *class* that inherits *instance methods*, *attributes*, and *instance variables* directly from another class, called the *parent class, base class*, *superclass,* or indirectly from an *ancestor class*. A subclass may also be called a *child class* or *derived class.*

subclassing The process whereby a new *class*, as it is created (or *derived*), inherits *instance methods, attributes,* and *instance variables* from one or more previously defined *ancestor classes.* The immediate *parent class(es)* of a new class must be specified in the class's *interface declaration.* See also *inheritance.*

superclass See *parent class.*

symbol In the Emitter Framework, any of a (standard or user-defined) set of names (such as, className) that are used as placeholders when building a text template to pattern the desired *emitter* output. When a template is emitted, the symbols are replaced with their corresponding values from the emitter's symbol table. Other symbols (such as, classSN) have values that are used by section-emitting methods to identify major sections of the template (which are correspondingly labeled as "classS" or by a user-defined name).

target object (Or *receiver.*) The object responding to a *method* call. The target object is always the first formal parameter of a *method procedure*. For SOM's C-language bindings, the target object is the first argument provided to the method invocation macro, *_methodName*.

usage bindings
The language-specific *binding* files for a *class* that are generated by the *SOM Compiler* for inclusion in client programs using the class.

value logging In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the new value of the object is distributed after the execution of a method that updates the object.

view–data paradigm
A Replication Framework construct similar to the Model-View-Controller paradigm in SmallTalk. The "view" object contains only presentation-specific information, while the "data" object contains the *state* of the application. The "view" and "data" are connected by means of an "observation" protocol that lets the "view" be notified whenever the "data" changes.

writers See *readers and writers.*

Index

A

B

C

D

E

K

L

M

N

O

T

U

V

W