Chapter 9. The Replication Framework

Contents

Chapter 9. The Replication Framework

9.1 Introduction

The continuing integration of electronic computation and electronic communication has generated a new kind of application. Whether called “a multiparty application,” “groupware,” or “computer-supported cooperative work,” this kind of application includes a dimension well beyond that of traditional distributed computing. Rather than merely placing a user in contact with a remote database (as in transaction processing — a common experience when people use an automatic teller machine), this new kind of application places a person in direct contact with other people.

Currently, the most common multiparty applications consist of two-player games where the players are located in the same room using one computer. In the future, however, multiparty applications will commonly coordinate more than two people who are not necessarily located together. For example, there are a number of group editors in the computer science literature. In addition, decision support systems for meeting rooms are beginning to emerge.

An essential aspect of successful multiparty applications is that participants must really feel they are touching one another. To facilitate the implementation of an application that meets this requirement, the SOMobjects Toolkit provides the **Replication Framework**.

The Replication Framework enables an object to be replicated (copied) in the address spaces of several processes distributed about a network. This provides the fast response times necessary for cooperative interactive applications. Each *replica* of the object can be updated, and the framework guarantees that the updates are serialized. (That is, there is a specific order in which updates will be performed, despite the fact that the updates originate from different processes.)

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 Although these methods have been superseded by the **somDefaultInit** and **somDestruct** methods, which are more efficient, be assured that **somInit** still executes correctly.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

9.2 Principles of the Replication Framework

Figure 1 depicts a simple but typical situation using the Replication Framework. Notice that there are multiple processes running on possibly different nodes of the network, and each process has a copy of a replicated object. Here, the object *Lassie* is an instance of the class “ReplicatedDog”, which is a subclass of “Dog”. “ReplicatedDog” obtains its ability to replicate objects by being a subclass of the SOM-provided class **SOMRReplicbl**.

The Replication Framework provides the following properties:

1. The number of replicas and their location in the network is hidden from the application program.
2. Updates (changes in value for instance variables) are communicated from the originating process to the other participants without the use of secondary storage.
3. The updates are serialized by the framework.
4. A participant is free at any time to join or leave the group of processes holding a replica.
5. The framework is tolerant of a single failure resulting from either a process crash or a network partition.

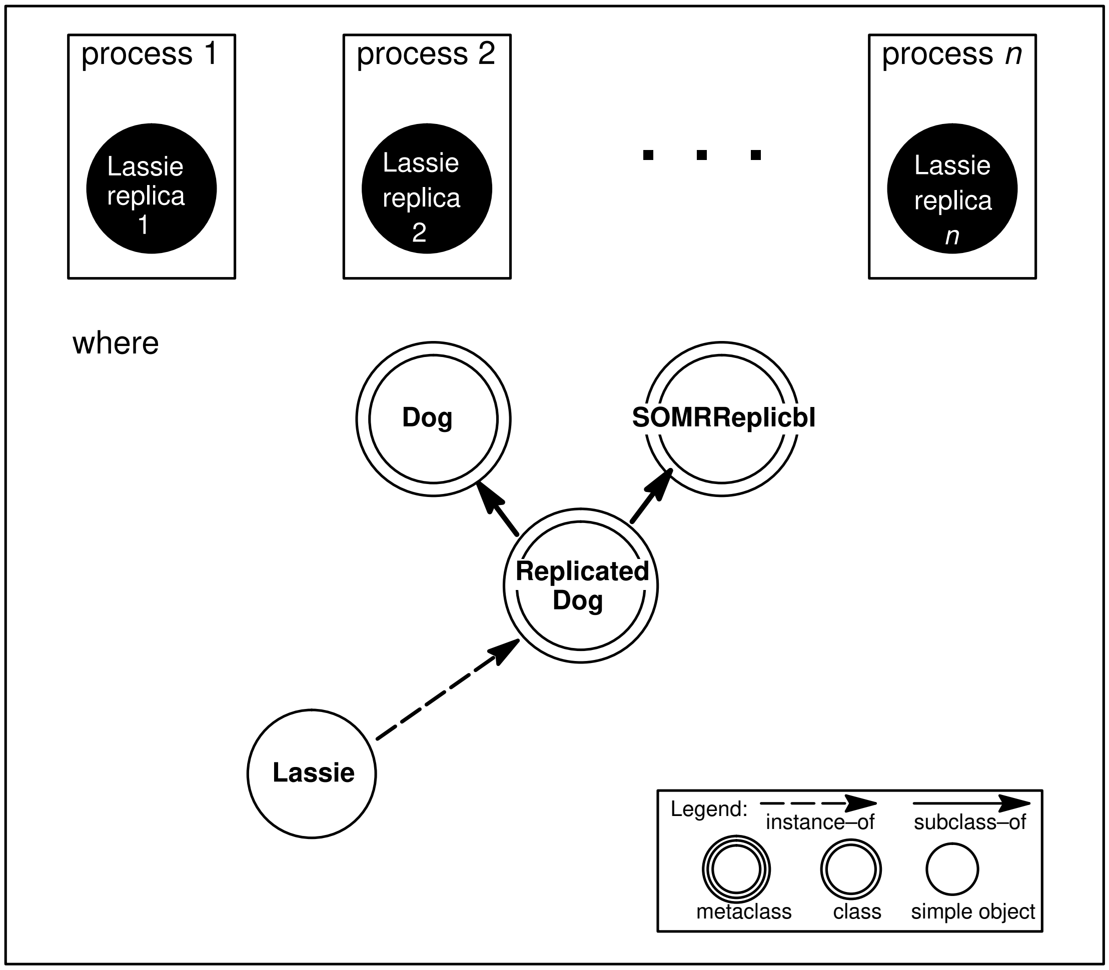

Figure 1. Multiple processes running replicas of a simple object.

Steps in using Replication

The Replication Framework can be exploited only if the applications are structured appropriately. The required structure is similar to the “Model–View–Controller” paradigm used by Smalltalk programmers. The Replication Framework proposes a View–Data paradigm. The Data object has whatever ‘state’ information the application desires to store in it. The View object has no state. It has methods to show a rendition of the state contained in the Data object. It may have some data that purely pertains to the image being displayed to the user. For example, in a visual presentation, the colors used for different regions may be in the View object, but the content information comes from the Data object.

Further, the View and Data must have a protocol between them such that whenever the Data object changes, a “signal” is sent to the View object to take note of the change and refresh the display (if necessary). This protocol can be extended to multiple Views on the same Data object, whereby an update to the Data object is automatically seen in all visual presentations. Effectively, the Views “observe” the Data.

The Replication Framework is concerned with Data objects only. It is the responsibility of the application developer to implement the “observation” protocol between the Views and Data. The Replication Framework requires the Data objects to be derived from a distinguished framework class **SOMRReplicbl.**

The recommended steps in building a replicated application are as follows:

- Structure the application to follow the View–Data paradigm, build the View and Data as **SOMObject**s, build a driver main program, and test the application as a stand-alone program.
- Convert the Data object (it is possible to have multiple replicated objects in a given program) to be a replicated object by making it a subclass of **SOMRReplicbl**. Do all the necessary method overrides to be a subclass of **SOMRReplicbl**. Again, test it as a stand-alone program.
- Enable replication by doing the appropriate replication initialization in the application’s main program, and test it with multiple copies of the application running.

9.3 Components of the Framework

The Replication Framework consists of three components, the:

- **SOMRReplicbl** class,
- **SOMR** class, and
- **.scf** file.

The **SOMRReplicbl** class is used to make subclasses that can create replicated objects. In Figure 1, for example, the class “ReplicatedDog” (a subclass of “Dog”) obtains its ability to replicate the object “Lassie” by also being a subclass of **SOMRReplicbl**. Techniques for making a “replicated” class (that is, a class that can create replicated objects) are discussed in the subsequent paragraphs.

The **SOMR** class provides a number of services for replicated objects. It is not necessary to know what they are; it is only necessary to create one instance of **SOMR** during the initialization of your program. [This is one of only two requirements imposed by the Replication Framework on the main program. The other one is the use of the Event Management Framework (see Section 9.7 for an example program).]

All replicated objects have names; it is this name by which the replicas locate each other on the network. That is, all objects initialized for replication under the same name (given during object initialization) are replicas of each other. Note that replicas must be instances of the same class, and that the name of the object is a null-terminated string.

Like all distributed systems, the Replication Framework needs a way to establish communication among the various participating processes. This is accomplished with the *.scf* files. (*scf* stands for share control file.) By convention, the *.scf* file corresponding to a set of replicas is named by appending “.scf” to the object’s name. Because the object’s name is just a string, the name may represent a path in the file system. If the name is a path, all the directories in the path must exist.

For example, in Figure 1 the processes initially locate each other by reading the “Lassie.scf” file. The Replication Framework creates such files automatically in the distributed file system. (Note: This is not a contradiction to the earlier statement that updates are communicated without the use of secondary storage. The *.scf* file is only used by replicas to establish communication channels among them. Once they are established, the updates are propagated on these channels without the help of secondary storage.)

Important: The *.scf* files are an artifact of the implementation that are visible to the user but are *not* part of the interface. As such, IBM reserves the right to eliminate them in the future. Do *not* make any programming decisions based on their content or even on their existence.

There are two ways in which changes can be propagated among replicas: *operation logging* and *value logging*:

- In operation logging, each method invocation that modifies a replica also executes at the site of the other replicas.
- In value logging, the change in value of a replica is encoded after a method invocation, and subsequently the values of the other replicas are upgraded to reflect this change.

The following two subsections give examples for subclassing an existing class to create a “replicated” class (a class capable of creating replicated objects) that uses either operation logging or value logging. Each of these subsections explains a recipe for creating either an operation-logged or a value-logged replicable class. If an application has no special needs, you can use the **SOMRReplicable** metaclass to create a replicable object instead of using the recipe. The **SOMRReplicable** metaclass is described in Chapter 10, “The Metaclass Framework.”

9.4 Making a “replicated” Class that provides Operation Logging

Following are the steps used to create a “replicated” class (for example, to create the class “ReplicatedDog” from the parent class “Dog”) whose replicated instances will be updated using operation logging. This example shows the C implementation; the C++ implementation is done similarly.

Step 1. Create a new class whose parents are both **SOMRReplicbl** and the original class (here, “Dog”) whose instances are to be replicated. Alternatively, you can directly derive “ReplicatedDog” from **SOMRReplicbl**.

Step 2. In the new class, override all the methods that change the value of instance variables (of “Dog”), override **somrDoDirective**, **somrGetState** and **somrSetState**. What to do in each overriding method is explained in the following steps.

Step 3. Each overriding method from the original class (here, “Dog”) should be coded as follows:

```
#include <somrerrd.h>

dogMethod( ReplicatedDog somSelf, <parameters> ) {
ReplicatedDogData *somThis = ReplicatedDogGetData(somSelf );
Environment *Env = SOM_CreateLocalEnvironment();
_somrLockNlogOp( somSelf,
                  Env,
                 ”ReplicatedDog”,
                 ”dogMethod”,
                 <parameters> );
if (Env->_major == NO_EXCEPTION ) {
      parent_dogMethod( somSelf, <parameters> );
      ... <any other additional code> ...
      _somrReleaseNPropagateOperation( somSelf, Env );
      }
else {
      /* code to handle failure to obtain a lock */
      switch (somriGetErrorCode(Env)){
      case SOMR_MASTER_UNREACHABLE:  ...
      case SOMR_UNAUTHORIZED:  ...
      case SOMR_TIMEOUT:  ...
      case SOMR_TRYLATER  ...
      default:  ...
      }
}
```

where `<parameters>` represents the explicit formal parameters of “dogMethod”. Note that for IDL call style the environment is listed among the <parameters> to **somrLockNLogOp**. The parameters to **somrLockNlogOp** must be one of the following types:

- one of the following primitive types: short, long, unsigned short, unsigned long, float, double, char, octet, boolean, and string.
- a pointer to a primitive type, or
- an object pointer.

If an object parameter is used, it must be an instance of one of the following classes:

- **SOMRReplicbl**
- **SOMPPersistentObject**
- **SOMRLinearizable**

There is an essential difficulty for the Replication Framework when using operation logging with methods that have objects as parameters — that is, the problem of how to repeat the operation in another address space where the object (to which the parameter refers) may not exist. The preceding restriction on object parameters deals with this difficulty in one of three ways:

- If the object is **SOMRReplicbl**, it is assumed that the object is already in the other address space and only the name of the object is passed.
- If the object is **SOMPPersistent**, the object name (used by the Persistence Framework) is passed and the object is restored (activated), if necessary.
- If the object is **SOMRLinearizable**, the object is copied to the other address space. A linearizable object must support the methods **somrGetState** and **somrSetState**; the class **SOMRLinearizable** contains the specification of both of these methods.

Passing any other object as a parameter to **somrLockNlogOp** can produce unpredictable results. If this constraint cannot be satisfied, then the application program should use value logging rather than operation logging to achieve replication, as described in the next section.

Note: If the “ReplicatedDog” class uses “callstyle = idl” in its interface definition, then the first parameter in <parameters> is always an Environment parameter, which is neither a primitive CORBA type nor an object of the right class. Currently the Replication Framework ignores the Environment parameter. However, it must be included in the parameter list <parameters> in order to match the number of arguments expected for the method. Also, currently “Context” object parameters are not supported. These restrictions on Environment and Context parameters apply only to those methods that need to be replicated.

Step 4. Initialize the object (replica) for replicated operation as follows (either in the main program or from inside of **somInit** of “ReplicatedDog”, if **somInit** is overridden):

```
#include <somrerrd.h>

ReplicatedDog dog
Environment *Env;
int rc;
dog = ReplicatedDogNew();
Env = SOM_CreateLocalEnvironment();

  _somrSetObjName( dog, Env, ”Lassie” );
  rc = _somrRepInit( dog, Env,  ’o’, ’w’);
  if (Env->_major == NO_EXCEPTION) {
     somPrintf(
       ”Successfully initialized for replication. rc = %d\n”,
                                                          rc);
        ...
       }
  else {
      somPrintf(”Initialization for replication failed\n”);
      switch(somriGetErrorCode(Env)) {
          case SOMR_MASTER_UNREACHABLE:  ...
          case SOMR_UNAUTHORIZED:  ...
          case SOMR_TRYLATER: ...
          default: ...
      }
```

The **somrRepInit** parameter ‘o’ indicates operation logging, while ‘w’ indicates this replica is a writer.

Step 5. Override **somrGetState** and **somrSetState** to format the state of the object as a byte string and to set the state of the object from such a string. Both methods require that the first four bytes of the string contain its length in binary. (Note that for a persistent object, you might use the Encoder/Decoder of the Persistence Framework, described in the previous chapter, with a Media Interface that returns a buffer.) The Replication Framework calls these methods on the replicated object to synchronize new replicas with the existing ones (**somrGetState** on existing replicas and **somrSetState** on new replicas). For help in implementing these two methods, refer to the sample programs shipped with the SOMobjects Developer Toolkit.

Step 6. To have a replicated object respond to *directives*, override the **somrDoDirective** method. See the subsequent section entitled “Handling Directives.”

9.5 Making a “replicated” Class that provides Value Logging

This example is similar to the previous one, except that this new class will use value logging to propagate updates among its replicas.

Step 1. Create a new class whose parents are both **SOMRReplicbl** and the original class (“Dog”) whose instances are like those that will be replicated.

Step 2. In the new class, override all the methods that change the value of instance variables (of “Dog”) and override the following methods of **SOMRReplicbl**: **somInit**, **somrApplyUpdates, somrDoDirective**, **somrGetState** and **somrSetState**. What to do in each overriding method is explained in the following steps.

Step 3. Each overridden method from the original class (such as “Dog”) should be coded as follows:

```
dogMethod( ReplicatedDog somSelf, <parameters> ) {
      char *buf;
      Environment *Env = SOM_CreateLocalEnvironment();
      _somrLock( somSelf, Env );
      if (Env->_major == NO_EXCEPTION) {
            parent_dogMethod( somSelf, <parameters> )
            buf = <some algorithm to capture the change
                        in the state of the object>;
            _somrReleaseNPropagateUpdate( somSelf
                                    Env,
                                    ”ReplicatedDog”,
                                    buf,
                                    <buf length in bytes>,
                                    0 );
            }
      else {
            /* code to handle failure to obtain a lock */
            switch (somriGetErrorCode(Env)){
            case SOMR_MASTER_UNREACHABLE:  ...
            case SOMR_UNAUTHORIZED:  ...
            case SOMR_TIMEOUT:  ...
            case SOMR_TRYLATER  ...
            default:  ...
            }
}
```

where `<parameters>` represents the explicit formal parameters of “dogMethod”.

Note that **somrGetState** may be used to capture the value of the object. However, you could propagate updates by differentials in the object value. This point reveals the subtle difference between value logging and merely capturing the value; that is, with value logging you have the possibility of computing and propagating differentials.

Step 4. Override **somInit** to initialize the object for replicated operation as follows:

```
SOM_Scope void  SOMLINK somInit(  ReplicatedDog somSelf)
{
    ReplicatedDogData *somThis = ReplicatedDogGetData( somSelf );
     /* make somInit calls on all parents */
    _somrSetObjName ( somSelf, Env, ”Lassie”);
    _somrRepInit ( somSelf, Env, ’v’, ’w’ );
}
```

The **somrRepInit** parameter ‘v’ indicates value logging, whereas ‘w’ indicates this replica is a writer.

Note that if the application will create more that one distinct instance of “ReplicatedDog” (for example, other names besides “Lassie”), this initialization cannot be done by overriding **somInit**, because there is no way to pass the different object names to the **somInit** method. Instead, the methods **somrSetObjName** and **somrRepInit** must be called separately after the replica is created (for instance, after the call to “ReplicatedDogNew”) as follows:

```
varLassie = ReplicatedDogNew();
_somrSetObjName ( varLassie, Env, ”Lassie”);
_somrRepInit ( varLassie, Env, ’o’, ’w’ );
```

Step 5. Override **somrGetState** and **somrSetState** to format the state of the object as a byte string and to set the state of the object from such a string. Both methods require that the first four bytes of the string contain its length. (Note that for a persistent object, you might use the Encoder/Decoder of the Persistence Framework (described in the previous chapter) with a Media Interface that returns a buffer.)

Step 6. To have a replicated object respond to *directives*, override the **somrDoDirective** method. See the subsequent section entitled “Handling Directives.”

9.6 Handling Directives

A *directive* is a message from the Replication Framework to a replica (or more specifically, to the application using the replica). A directive indicates that some condition has arisen asynchronously (not as a reaction to any request by the local replica). The **somrDoDirective** method is provided by the framework to interpret directives sent to a replica. To enable replicated objects to handle directives appropriately for the application, override the **somrDoDirective** method. The overriding method might be coded as follows:

```
SOM_Scope void SOMLINK somrDoDirective( ReplicatedDog somSelf,
                                        Environment *env,
                                        string directive )
{
        if ( !strcmp( directive, ”BECOME_STAND_ALONE” ) {
                ... }
        else if ( !strcmp( directive, ”CONNECTION_LOST” ) {
                ... }
        else if ( !strcmp( directive, ”CONNECTION_REESTABLISHED” ) {
                ... }
}
```

where the ellipses represent application-dependent code. Currently defined directives are as follows:

BECOME_STAND_ALONE	— The replica has lost its connection to other replicas and the Replication Framework has given up trying to reconnect to the set of replicas. An application may subsequently try to reconnect by invoking first **somrRepUninit** and then **somrRepInit**.
LOST_CONNECTION	— The replica has lost its connection to the other replicas, but the Replication Framework is trying to reconnect; the directive below will be issued after connection is reestablished. You should not attempt to update the object at this time.
CONNECTION_REESTABLISHED	— The connection between the replica and its cohorts has been reestablished; processing may continue normally.
LOST_RECOVERABILITY	— This directive is issued by the framework when the *.scf* file cannot be updated to reflect the current state of the framework (due to network or file errors). Hence, because the framework uses the *.scf* files for recovery, recovery may be impacted adversely.

Figure 2 depicts the states of a replica and how transitions are made. Note that from the stand-alone state, **somrRepUninit** should be invoked before trying to invoke **somrRepInit** and that neither **somrRepInit** nor **somrRepUninit** should be invoked inside of the method **somrDoDirective** (because doing so deletes the object on which **somrDoDirective** is invoked). In addition, bear in mind that the **BECOME_STAND_ALONE** directive is received because something is amiss out in the network; it is prudent to wait a bit before trying to reconnect.

Note that in the **stand-alone** and **isolated** states the methods **somrLockNLogOp** and **somrLock** cannot succeed. Attempts to invoke these methods yield a return code of **SOMR_TRYLATER.**

Because of their passive nature, read-only replicas do not receive **BECOME_STAND_ALONE** directives, but can receive **LOST_CONNECTION** directives.

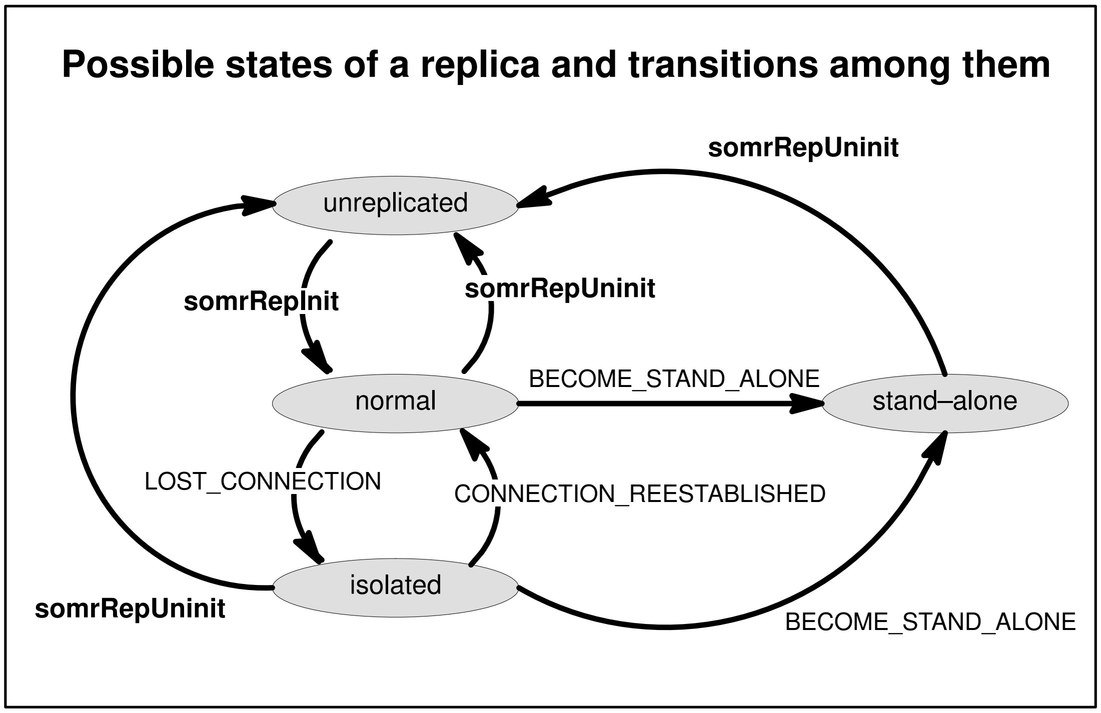

Figure 2. Possible states of a replica and transitions among them.

9.7 Repdraw: A Complete Example

This section presents a complete program using the Replication Framework (implemented in C). This program, called *repdraw*, enables a group of people to share a tablet on which they may draw simultaneously. This program is based on the program "xmdraw.c" in the *IBM AIX Windows Programming Guide* by Ian J. Charters (IBM UK) and Mark Connell (IBM Australia).

This example does the following two things:

- First, the drawing surface is made into a SOM object (an instance of the "tablet" class), and
- Second, each user will receive a replica of the "tablet" object.

Operation logging is illustrated in this example.

For brevity, the example creates a replicated object directly (rather than subclassing an existing class, as described earlier). Figure 3 contains the main program. Figure 4 contains the .idl file for the "tablet" class, and Figure 5 contains the implementation of the "tablet" class.

The following notes explain certain statements in the program. The number corresponding to each note also appears along the right margin of the code at the relevant point.

Note 1. This statement includes the header file, "somr.h", for initializing the Replication Framework. (C++ users should include "somr.xh".)

Note 2. This statement includes the header file, "eman.h", for initializing the Event Management Framework. (C++ users should include "eman.xh".)

Note 3. This statement creates and initializes the Replication Framework.

Note 4. This statement creates and initializes the Event Management Framework.

Note 5. This statement registers the callback for X events with the Event Management Framework.

Note 6. This statement flushes the X events before turning control over to the Event Manager.

Note 7. This statement turns over control to the Event Manager(EMan).

Note 8. Because **SOMRReplicbl** is a parent of the "tablet" class, file "tablet.ih" includes the appropriate header files for implementing a replicated class.

Note 9. In this simple example, no action is taken if the process fails to get a lock.

Note 10. In this simple example, no attempt is made to recover from an unsuccessful return from initialization for replication.

Main program for “repdraw” (first part)

```
#include <tablet.h>
#include <somr.h>                                                        /*01*/
#include <eman.h>                                                        /*02*/

void begin (Widget, caddr_t, XEvent *);
void paint (Widget, caddr_t, XEvent *);
void ProcessXEvent (SOMEEvent ,void *);
void RegisterXfd  (SOMEEMan);

tablet       myTablet;
Widget       toplevel;
XEvent     *tabletEvent;

main(int argc, char **argv, char **envp)
{
    Widget       drawarea;
    GC           myGC;
    int          n = 0;
    Arg          args[10];
    XGCValues    val;
    SOMR         myOwnSelf;
    SOMEEMan     EManPtr;
    Environment *tabletEnv = somGetGlobalEnvironment();

    myOwnSelf = SOMRNew();                                               /*03*/
    EManPtr = SOMEEManNew();                                             /*04*/
    toplevel = XtInitialize(argv[0],"RepDraw", NULL,0, &argc, argv);
    XtSetArg (args[n], XmNwidth,  500); n++;
    XtSetArg (args[n], XmNheight, 500); n++;
    drawarea = XmCreateDrawingArea (toplevel, "DrawArea", args, n);
    XtManageChild (drawarea);
    XtAddEventHandler (drawarea, ButtonPressMask,  False, begin, NULL);
    XtAddEventHandler (drawarea, ButtonMotionMask, False, paint, NULL);

    XtRealizeWidget ((Widget)toplevel);

    n = 0;
    XtSetArg(args[n], XmNforeground, &val.foreground); n++;
    XtSetArg(args[n], XmNbackground, &val.background); n++;
    XtGetValues(drawarea, args, n);
    val.foreground = val.foreground ^ val.background;
    val.function = GXxor;
    myGC = XtGetGC (drawarea, GCForeground|GCBackground|GCFunction, &val);
    myTablet = tabletNew();
    _setTablet (myTablet, drawarea, myGC);

    RegisterXfd(EManPtr);                                                /*05*/
    ProcessXEvent(NULL, NULL);                                           /*06*/
    _someProcessEvents(EManPtr, tabletEnv);                              /*07*/
}
```

Figure 3. The main program for “repdraw” (first part)

Main program for “repdraw” (second part)

```
/* =========================== Event Handlers ===========================
*/

void begin (Widget w, caddr_t client_data, XEvent *event)
{
    _beginLine (myTablet, event->xbutton.x, event->xbutton.y);
}

void paint (Widget w, caddr_t client_data, XEvent *event)
{
    switch (event->xmotion.state) {
    case Button1MotionMask:
         _drawLine (myTablet, event->xbutton.x, event->xbutton.y);
         break;
    case Button3MotionMask:
         _clearAll (myTablet);
         break;
    }
}

void ProcessXEvent(SOMEEvent foo, void *bar)
{
while (XPending (XtDisplay(toplevel))) {
         XtNextEvent(&tabletEvent);
         XtDispatchEvent(&tabletEvent);
    }
}

void RegisterXfd(SOMEEMan EManPtr)
{
SOMEEMRegisterData regData = SOMEEMRegisterDataNew();
Environment *tabletEnv = somGetGlobalEnvironment();
int Xfd;
    Xfd = XConnectionNumber(XtDisplay(toplevel));
    _someClearRegData(regData,tabletEnv);
    _someSetRegDataEventMask(regData, tabletEnv, EMSinkEvent, NULL);
    _someSetRegDataSink(regData, tabletEnv, Xfd);
    _someSetRegDataSinkMask(regData, tabletEnv, EMInputReadMask |
                                                 EMInputExceptMask);
    _someRegisterProc(EManPtr, tabletEnv, regData, ProcessXEvent, NULL);
}
```

Figure 3.   The main program for “repdraw” (second part)

IDL specification of the “tablet” class

```
#include <replicbl.idl>

interface tablet : SOMRReplicbl
{
  void setTablet( in void* clientDA, in void* clientGC );
  void beginLine( in long x, in long y );
  void drawLine( in long x, in long y );
  void clearAll();

#ifdef __SOMIDL__

implementation {
  callstyle = oidl;
  releaseorder: setTablet, beginLine, drawLine, clearAll;

  filestem = tablet;
  passthru C_h =
        ”#include <X11/Intrinsic.h>”
         ”#include <Xm/Xm.h>”
         ”#include <Xm/DrawingA.h>”;

  int x1;
  int y1;
  int x2;
  int y2;
  void *gc;
  void *da;

  somInit: override;

 };

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

Figure 4. The IDL specification of the “tablet” class.

Implementation of the “tablet” class (first part)

```
#define tablet_Class_Source
#include ”tablet.ih”                                                     /*08*/
Environment *tabletEnv;
SOM_Scope void SOMLINK setTablet(tablet somSelf,void *clientDA,void *clientGC)
{
    tabletData *somThis = tabletGetData(somSelf);
    _da = (Widget)clientDA; _gc = (GC)clientGC;
}
SOM_Scope void SOMLINK beginLine(tablet somSelf, long x, long y)
{
    char *buf;
    int  buflen;

    tabletData *somThis = tabletGetData(somSelf);
    _somrLockNlogOp(somSelf,tabletEnv,”tablet”,”beginLine”,x,y);         /*09*/
    if (tabletEnv->_major == NO_EXCEPTION) {
         _x1 = x; _y1 = y; _x2 = x; _y2 = y;
         _somrReleaseNPropagateOperation (somSelf, tabletEnv); }
}
SOM_Scope void SOMLINK drawLine(tablet somSelf, long x, long y)
{
    char *buf;
    int  buflen;

    tabletData *somThis = tabletGetData(somSelf);
    _somrLockNlogOp(somSelf,tabletEnv,”tablet”,”drawLine”,x,y);          /*09*/
    if (tabletEnv->_major == NO_EXCEPTION) {
         _x2 = x;  _y2 = y;
         XDrawLine(XtDisplay((Widget)_da), XtWindow((Widget)_da),
                   (GC)_gc, _x1, _y1, _x2, _y2);
         XDrawPoint (XtDisplay((Widget)_da), XtWindow((Widget)_da),
                   (GC)_gc, _x2, _y2);
         XFlush (XtDisplay((Widget)_da));
         _x1 = _x2;  _y1 = _y2;
         _somrReleaseNPropagateOperation(somSelf, tabletEnv);}
}
SOM_Scope void SOMLINK clearAll(tablet somSelf)
{
    char *buf;
    int  buflen;
    tabletData *somThis = tabletGetData(somSelf);
    _somrLockNlogOp( somSelf, ”tablet”, ”clearAll” );                    /*09*/
    if (tabletEnv->_major == NO_EXCEPTION) {
         XClearWindow ( XtDisplay((Widget)_da), XtWindow((Widget)_da) );
         XFlush (XtDisplay((Widget)_da));
         _somrReleaseNPropagateOperation (somSelf, tabletEnv); }
}
```

Figure 5.   Implementation of the “tablet” class (first part)

Implementation of the “tablet” class (second part)

```
SOM_Scope void  SOMLINK somInit(tablet somSelf)
{
    char *strname = ”tablet”;
    tabletData *somThis = tabletGetData(somSelf);
    parent_somInit (somSelf);
    tabletEnv = somGetGlobalEnvironment();
    _somrSetObjName (somSelf, tabletEnv, ”Tablet”);
    _somrRepInit (somSelf, tabletEnv, ’o’, ’w’);                       /*10*/
}
```

Figure 5.   Implementation of the “tablet” class.

9.8 Miscellaneous Considerations

Composition and nesting

The Replication Framework does compose with itself. That is, the situation depicted in Figure 6 does work, because the SOM model uses graph inheritance. The situation shown here is unusual, because it is unlikely that the implementors of classes “Collie” and “ReplicatedCollie” would not know that “ReplicatedDog” is derived from **SOMRReplicbl**. Beware: if the class “Collie” introduces additional update methods, they must be constructed according to the examples given earlier for making a class “replicated”.

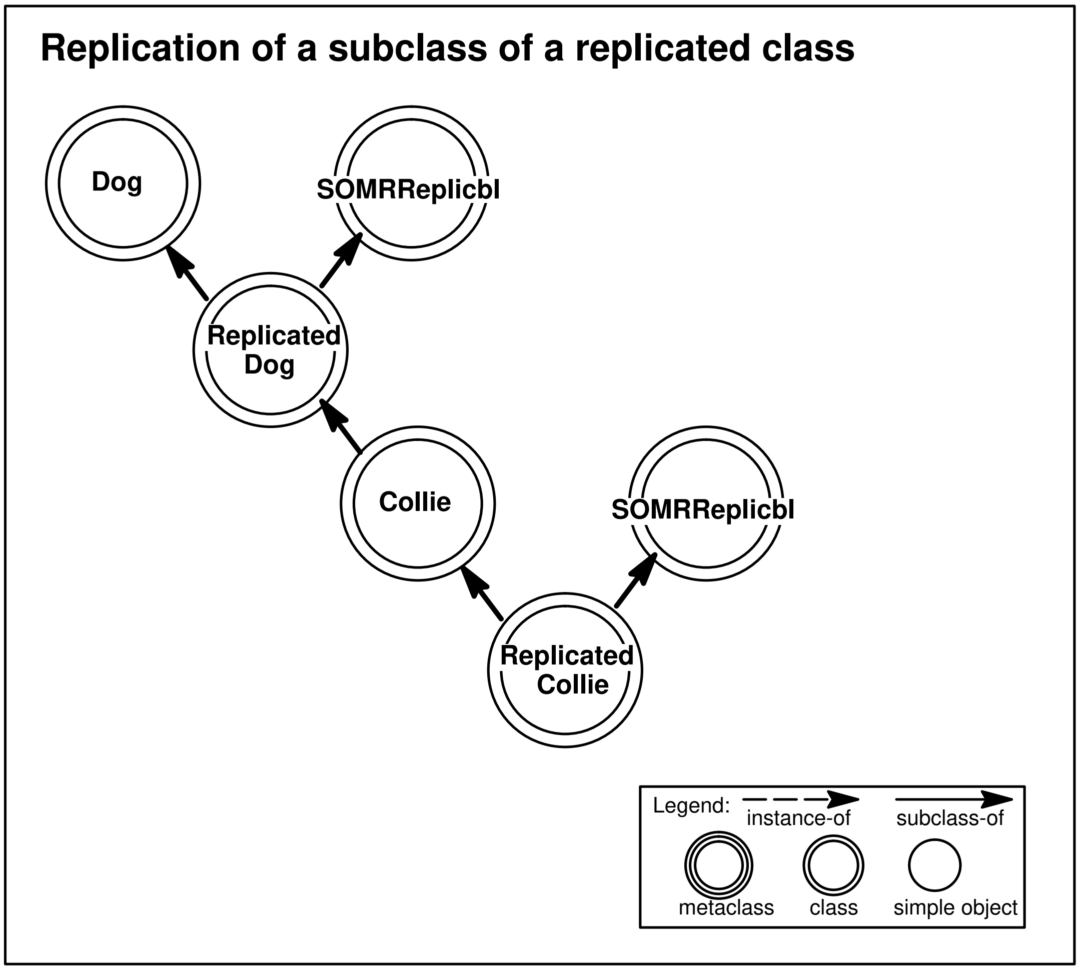

Figure 6. Replication of a subclass of a replicated class.

Furthermore, methods within a replicated object may call other such methods. That is, when doing operation logging, it does no harm to nest the method pair **somrLockNlogOp**/ **somrReleaseNPropagateOperation** with another such pair (because of a method invocation). A similar situation exists for value logging.

Choosing the type of logging

To operation log, or not to operation log — that is the question (for which there is no simple answer). Figure 7 depicts the basic heuristic for making this decision: The x axis represents the duration of the average update to a replicated object. The y axis represents the amount of change made by the average update (which might be measured in the number of bits that change in the object representation).

Operation logging has advantages when updates run quickly or make large changes to large objects. For long-running updates, it may be better to use value logging, which in effect transfers only the result of the operation to the replicas. On the other hand, for a very large object, transferring the result among the replicas might take longer than having the replicas repeat the execution of the method.

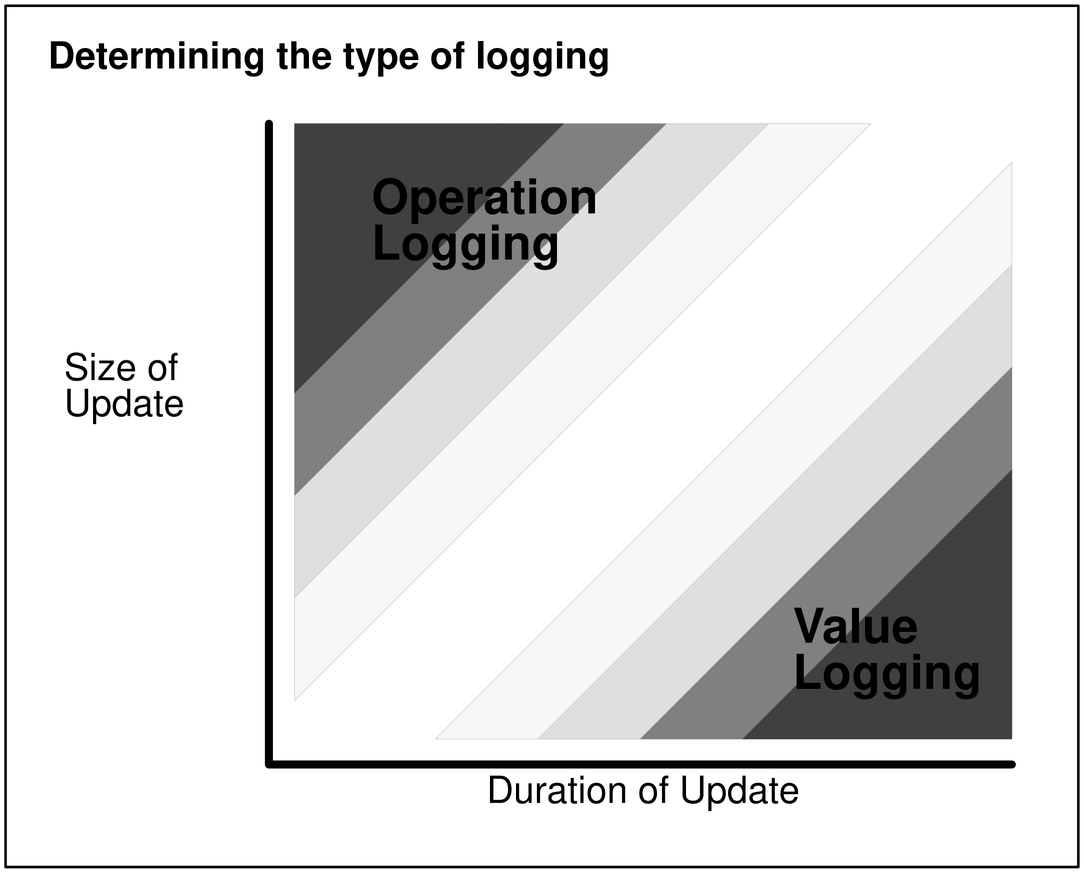

Figure 7. A heuristic on how to determine the type of logging.

Replication and databases

It is common to initialize a replicated object from a database. A problem arises, however, when considering how to update the database when the replicated object changes. This may not seem like a problem, but without a proper design all replicas could be updating the database. Consider a technique that avoids this problem: First, note that a method need not log its own name when invoking **somrLockNlogOp**. Thus, a user can separate a method to change a replicated object into two methods: one that is called by the application and one that is propagated and called by the Replication Framework. The first method might look like this:

```
changeReplicaAndDatabase( replicaType *somSelf, ... )
{
        somrLockNLogOp( somSelf, ”changeReplicaOnly”, ... );
        changeReplicaOnly( somSelf, ... );
        <update database from changed replica>;
        somrReleaseNPropagateOperation( somSelf );
}
```

In this way the database is updated only from the site of the replica where the change originated. Replicas that receive the change from the Replication Framework only modify the replica, not the database.

Aborting a method

Suppose it is necessary to abort a method. That is, after **somrLockNlogOp** (for operation logging) or **somrLock** (for value logging), it is not desirable to communicate the effects of the method to the other replicas. Before aborting the method, first restore the object to its state at the time that the lock was obtained. Then, to execute the abort, call **somrReleaseLockNAbortOp** (for operation logging) or **somrReleaseLockNAbortUpdate** (for value logging).

Readers and writers

The Replication Framework supports reader and writer replicas. The readers, as the name implies, should not attempt updates on the replicated object. If they do, they fail to get the replica lock. The *.scf* files mentioned in section 9.3 are also used to enforce the access control policy for replicated objects. For example, processes that have “write” access to the shared control file can become writer replicas, and processes with “read” access can only become reader replicas. Processes with no access to the *.scf* cannot participate in replication.

You can control the file access permission mask by overriding the **somrGetSecurityPolicy** method of **SOMRReplicbl** class. (The value returned by **somrGetSecurityPolicy** becomes the third parameter of a file “open”; the interpretation is operating-system dependent.) The Replication Framework checks the access permissions of a process (reader or writer) when a replica registers itself by calling **somrRepInit,** and subsequently enforces the appropriate access control. If a replica wants to change its mode from read to write or vice versa, it must do a **somrRepUninit** followed by a **somrRepInit** with the appropriate mode.

Storage considerations

When using operation logging, you should avoid logging operations that return dynamically allocated storage (this includes SOM objects). When such methods are invoked at replicas, return values are discarded; thus, any dynamically allocated storage is lost. This situation also applies to methods in which dynamically allocated storage is returned in parameters. This observation demonstrates that not all classes are equally amenable to operation logging. (The parameters to a replicated method are essentially passed ”by value” only. Any values returned through reference parameters are lost.)

If a replicated object uses CORBA style method calls (i.e., its class has the SOM IDL modifier callstyle=idl), then the exceptions returned through the environment parameter are ignored when the log is replayed at replicas.

Performance considerations

With regard to performance, consider the number of messages required to perform a single update of an object with *n* replicas: In the best case, *n*–1 messages are required. In the worst case, *n*+3 messages are required.

How to handle slow networks

The Replication Framework timeout is defined in the environment variable SOMR_RPCTIMEOUT. If the network is slow, this timeout may be exceeded, which in turn causes replicas to begin working without coordination with the others. If this happens, increase the timeout setting. Remember that time is given in milliseconds.

Fault-tolerance

The Replication Framework tolerates process crashes and network partitions. When the network partitions, the issue arises as to which subset of replicas will continue. For the Replication

Framework, it is the subset that still has access to the .scf file. Each replica in the other partition receives the BECOME_STAND_ALONE directive.

The case of process crashes is somewhat more complicated. The excellent performance of the framework is achieved by designating one replica as the *master* (with the other replicas called *shadows*). If a shadow crashes, the set of replicas continue to operate with no problem. If the master crashes, however, some time is required before the shadows recognize this and elect a new master.

The duration of this recovery is controlled by two environment variables: SOMR_HEARTBEAT and SOMR_INTERBEATLIMIT. These two are set as positive integers that represent times in milliseconds. The times control failure detection and recovery algorithms. SOMR_INTERBEATLIMIT must be greater than SOMR_HEARTBEAT (typically two or three times greater). The framework provides default values of 30 and 90 seconds. Setting SOMR_INTERBEATLIMIT too high lengthens the recovery period while deceasing the possibility of falsely declaring a process to be dead. Setting SOMR_HEARTBEAT too low can degrade performance.

In addition, SOMR_INTERBEATLIMIT and SOMR_HEARTBEAT should be set greater than the maximum difference of the clock readings of the computers on which the replicas are running. For example, if the clocks of the computers cannot be synchronized closer than two minutes do not set SOMR_INTERBEATLIMIT and SOMR_HEARTBEAT less than two minutes.

Thread safety

The Replication Framework does not provide thread safety. You should never call methods of the framework concurrently. If this is a problem in a multi-threaded application, consider constructing an atomic replicated object using Before/After Metaclasses (see Chapter 10).

Persistent objects

Because a replicated object may be updated in response to changes at other sites, you should take care when saving a replicated object that is also persistent. The recommended way is to lock the object (use **somrLockNlogOp** for operation logging or **somrLock** for value logging), save the object, and then abort the operation (with **somrReleaseLockNAbortOp** for operation logging or with **somrReleaseLockNAbortUpdate** for value logging).

Deadlock avoidance

Because replicas lock the replicated object (with **somrLock** and **somrLockNlogOp**), deadlocks are possible if methods that change an object invoke other methods on replicated objects. To avoid deadlocks, use any of the standard techniques to break circular waits. For example, impose a linear order on the objects and never lock a lower precedence object until all required higher precedence objects have been locked.

Environment variables

Following is a summary of the environment variables that control operation of the Replication Framework. The framework assumes default values if these variables are not set in the user environment.

SOMR_HEARTBEAT	— The time in milliseconds that some processes use to announce that they are still alive. The default value is 30 seconds.
SOMR_INTERBEATLIMIT	— The maximum time in milliseconds to wait for a heart beat (after which a process may be declared dead). The default value is 90 seconds.
SOMR_RPCTIMEOUT	— The time in milliseconds that is used to wait for the acknowledgement of a message. The default value is 30 seconds.

SOMR_SCFDIRECTORY	— A prefix that is added to object names to make a file name for the .scf file.
SOMR_SCFDURATION	— The maximum time, in milliseconds, that the .scf file is locked for a single operation. The Replication framework uses a default value. However, the framework can detect that this value is not appropriate (for example, in a slow file system) and may ask you to increase it. The default value is 3000.
SOMR_DOSNFS	— This environment variable is *only* applicable on a Windows platform, with the .scf files being created on a DOSNFS directory. In such an environment, set SOMR_DOSNFS=1 for error-free operation.
SOMSOCKETS	— The name of the socket implementation class used for a given application. This variable is usually fixed at the SOMobjects Toolkit installation time. (There is no default setting.)
SOMIR	— The path name(s) of the interface repository. It must include a definition of the socket implementation class indicated by SOMSOCKETS.
MALLOCTYPE=3.1	— This is only for AIX. It tells the malloc/free routines in the AIX standard C library to use the memory management algorithms from AIX Version 3.1, versus the algorithms used in Version 3.2. The later algorithms apparently sometimes cause problems. Setting MALLOCTYPE=3.1 usually fixes the problems the Replication Framework occasionally experiences with malloc/free on AIX 3.2.

Return codes

Given below are the codes returned by Replication Framework methods; the reference manual page on each method states which code a method may return. After a Replication Framework method call, these error codes can be accessed from the **Environment** parameter. For convenience, these codes are also listed in Appendix A, “Customer Support and Error Codes,” along with the codes of the entire SOMobjects Developer Toolkit. A brief list of possible actions is provided wherever appropriate.

Description	Value	Explanation
SOMR_TIMEOUT	500	— Possible actions are (1) to retry or (2) to terminate.
SOMR_OK	501	
SOMR_GRANTED	501	
SOMR_UNAUTHORIZED	502	— The likely cause is that the *.scf* file is inaccessible. Or it could be a reader trying to update the replicated object. Recovery action is to ensure proper access.
SOMR_TRYLATER	503	— Possible action is to wait for a while and retry the failed operation.
SOMR_DENIED	504	— The likely cause is that the *.scf* file is inaccessible. Or it could be a reader trying to update the replicated object. Recovery action is to ensure proper access.

Description	Value	Explanation
SOMR_MASTER_UNREACHABLE	508	— The likely cause is that either the network is down or too slow. Possible actions are (1) Change the time constants through environment variables mentioned earlier. (2) Wait for a while and retry. (3) Ensure that the *.scf* file is accessible.

Messages

It is possible to receive the following messages from the Replication Framework while an application is running. All but the last indicates a misuse of the framework interface or a timing problem.

Replication operation not logged. Probable invalid parameter.
Check the reference manual and rewrite program with appropriate parameters.

somrApplyUpdates in class SOMRReplicbl called. Method must be overridden.
You are using value logging but have not overridden **somrApplyUpdates**.

Warning: Trying to UnPin a replicated object that is not Pinned.
Each call to **somrUnPin** must be preceded by a call to **somrPin**; check your program.

Waiting for Network Transport to be ready...
This message usually appears when the communication buffers are full. When the target application consumes the pending messages, the problem goes away. Occasionally, this can also happen due to a programming error (for example, if a process containing a replica that is the target of update messages blocks indefinitely or enters an infinite loop).

Shutting down listener until some replicas terminate.
This message indicates that the number of replicas reached the permitted maximum.

Listening to connections again.

SOMRERROR Replication Framework Error: N.N.N. Refer to IBM Customer Service.
This message is issued by an internal consistency check in the framework and should never appear. Because of the fault-tolerance of the framework, your application may continue to run correctly. However, the message should be reported so that IBM can provide improvements to the framework.

Environment variable SOMSOCKETS is not defined.
See the subsection “Dependence on Sockets DLL” in this chapter.

Unable to locate the class <class name> in SOMIR or failed to load the associated dll.
The specified class name is either not found in the implementation repository (indicated by the environment variable SOMIR) or the corresponding dynamic load library could not be found.

Operating system considerations

Although the Replication Framework tries to present the same interface for all platforms, this is not possible in one circumstance — the manner in which the main loop of an application is written. This is because each operating system provides different primitives for process control (for example, currently OS/2 has threads while AIX and Windows do not). Sample code is provided to demonstrate applications for all operating systems; you should study them carefully.

Dependence on sockets DLL

The Replication Framework uses sockets for inter-process communication. It also uses the services of EMan (of the Event Management Framework), which also uses sockets. The SOMobjects Toolkit abstracts socket facilities through the **Sockets** class. There are multiple concrete implementations of this class. For example, **TCPIPSockets** class uses the underlying TCPIP implementation. Both the Replication Framework and the Event Management Framework must be bound to a concrete implementation for any application to run.

Each socket implementation is provided in a separate DLL. The interface repository contains the information regarding which DLL contains the implementation of a given class. You can bind a concrete socket implementation to your application by setting the environment variable SOMSOCKETS to name the appropriate implementation class. Current choices are **TCPIPSockets, NBSockets,** and **IPXSockets**. Alternatively, you can specify your own socket implementation class that supports the abstract interface defined in **Sockets** class. (See Appendix E for a description of the **Sockets** class and requirements for implementing your own.)

Note that regardless of whether replication is used among processes on a single workstation or processes on a network of workstations, there must be a socket implementation. The availability of particular Socket implementation subclasses and their transport prerequisites differ depending on the SOMobjects package you have purchased.

Client events used by the framework

The Replication Framework uses clients events of the Event Management Framework. If an application also intends to use client events, then care must be taken to avoid name collisions with respect to client event types. The Replication Framework uses a client event type named “deferred”.

Writing X/MOTIF applications

When the Replication Framework is used in an X/MOTIF application, the subsequent instructions must be followed.

1. The MOTIF toolkit must be initialized (for example, via XtInitialize) and the X connection number obtained using the macro “ConnectionNumber” or a function call to “XConnectionNumber”. This connection number should then be registered with EMan (see Chapter 12, “The Event Management Framework”) with a callback method/procedure for X events. In the callback, you can process X events appropriately. After all necessary registrations are complete, control must be turned over to EMan by calling **someProcessEvents.**

2. Before turning control over to **someProcessEvents**, the callback for X events must be called once to process all the X events generated until then.

3. In the callback, you should process all available X events, rather than processing just one (that is, loop until XPending returns zero).

4. Inside of each replicated method that updates data objects that are being displayed, a call to XFlush must be made to refresh the display. Without this call to XFlush, the display may lag behind the actual state of the data objects.

All three points above are illustrated in the sample program shown previously in section 9.7, “Repdraw: A complete example.”

Tips on using Replication

The following are some do's and don'ts for the Replication Framework:

- Design your applications with clear view–data separation.
- Design your data objects so that they can work with multiple views (that is, multiple views can "observe" them).
- Any application using replication must use the Event Manager (EMan). (Refer to "Tips on using EMan" in Chapter 12, "The Event Management Framework.")
- Don't have very large composite objects as a single replicated object. This may be easier to program, but it leads to very coarse replica locks and hence poor performance.
- Replicated-object methods that acquire and release replica locks should not block for long periods of time.
- While designing applications with multiple replicated objects making method calls among them, ensure that there are no circular dependencies. Otherwise, deadlocks are possible.
- Do check for error return codes from replica lock calls. Proceeding with the update and calling the **somrReleaseNPropagateUpdate, somrReleaseNPropagateOperation, somrReleaseLockNAbortOp,** or **somrReleaseLockNAbortUpdate** method without successfully obtaining the replica lock can produce unpredictable results.
- Do not attempt a second **somrRepInit** call on a given replicated object without doing a **somrRepUninit** to nullify the first call. Doing so may terminate the program.

9.9 Limitations

Check the file "README" in the SOM root directory for limitations of the Replication Framework.

Chapter 10. The Metaclass Framework

Contents

Chapter 10. The Metaclass Framework

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

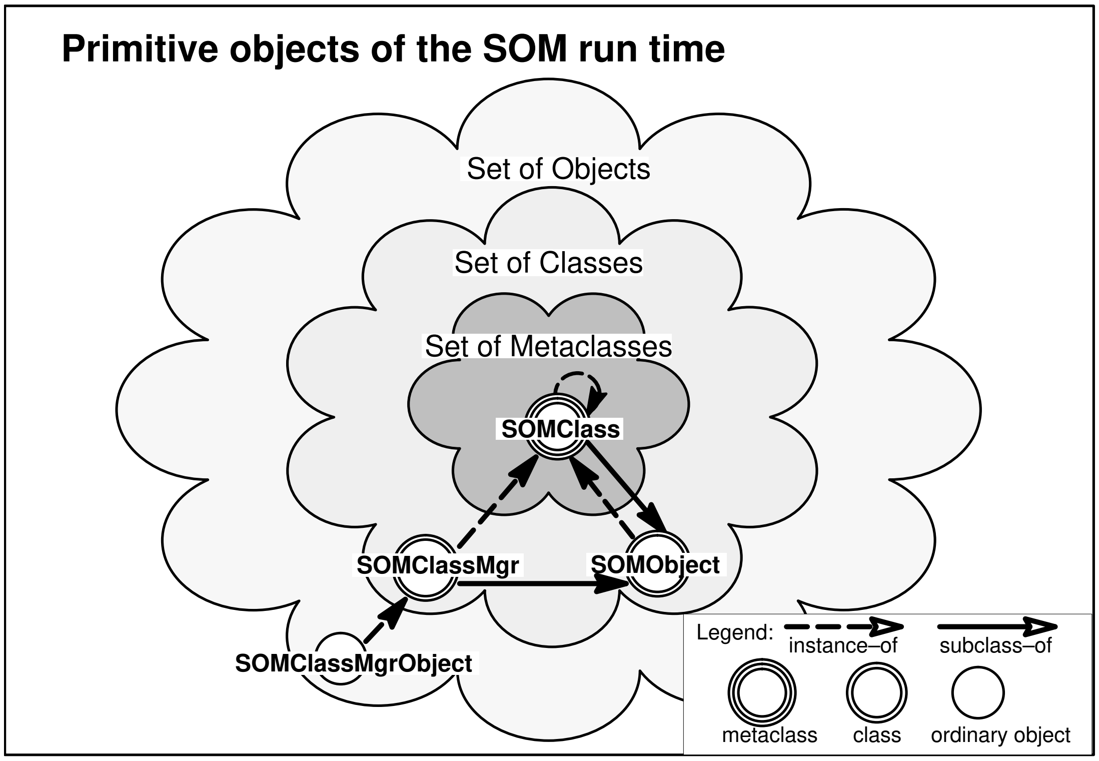

Figure 1.   The primitive objects of the SOM run time.

The important point to observe here is that any class that is a *subclass* of **SOMClass** is a *metaclass*. This chapter describes metaclasses that are available in SOMobjects Toolkit. There are two kinds of metaclasses:

Framework metaclasses — metaclasses for building new metaclasses, and
Utility metaclasses — metaclasses to help you write applications.

Briefly, the SOMobjects Toolkit provides the following metaclasses of each category for use by programmers:

- *Framework metaclasses:*

SOMMBeforeAfter	— Used to create a metaclass that has “before” and “after” methods for *all* methods (inherited or introduced) invoked on instances of its classes.

- *Utility metaclasses:*

SOMMSingleInstance	— Used to create a class that may have at most one instance.
SOMMTraced	— Provides tracing for every invocation of all methods on instances of its classes.
SOMRReplicableObject	— Provides an encapsulation of the Replication Framework (see Chapter 9).
SOMRReplicable	— Provides the Before/After method required by SOMRReplicableObject.

The diagram in Figure 2 depicts the relationship of these metaclasses to **SOMClass** (for completeness, the figure includes the metaclasses that are derived). The following sections describe each metaclass more fully. The ellipses indicate that there are additional metaclasses being used that are not part of the public interface.

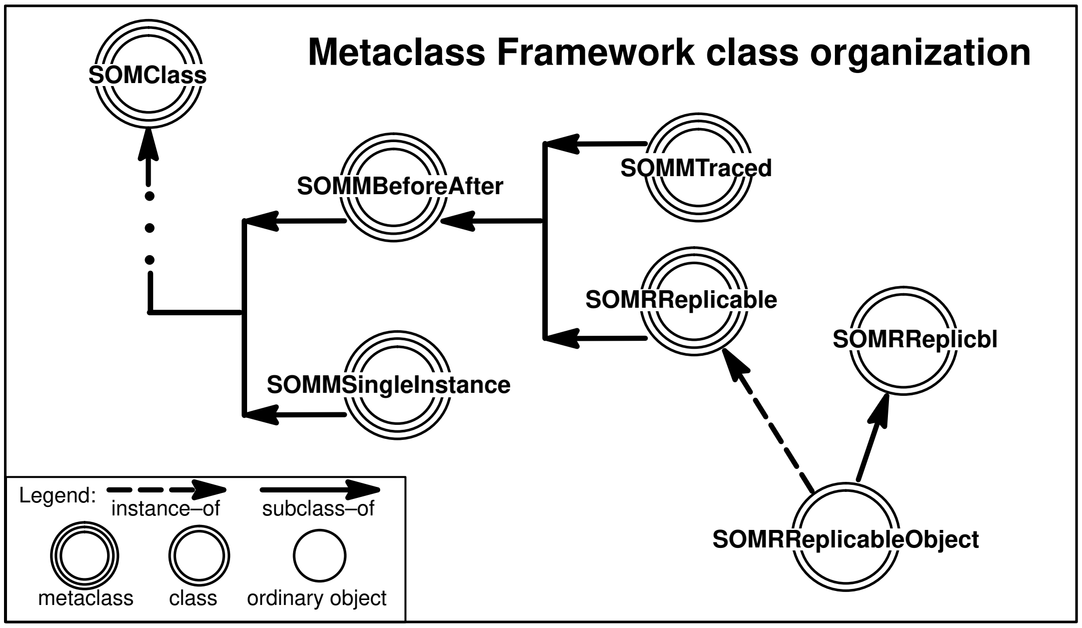

Figure 2. Class organization of the Metaclass Framework.

A note about metaclass programming

SOM metaclasses are carefully constructed so that they compose (see Section 10.1 below). If you need to create a metaclass, you can introduce new class methods, and new class variables, but you should not override any of the methods introduced by **SOMClass**. If you need more than this, request access to the experimental Cooperation Framework used to implement the Metaclass Framework metaclasses described in this chapter.

10.1 Framework Metaclasses for "Before/After" Behavior

The 'SOMMBeforeAfter' metaclass

SOMMBeforeAfter is a metaclass that allows the user to create a class for which a particular method is invoked *before* each invocation of every method, and for which a second method is invoked *after* each invocation. **SOMMBeforeAfter** defines two methods: **sommBeforeMethod** and **sommAfterMethod**. These two methods are intended to be overridden in the child of **SOMMBeforeAfter** to define the particular "before" and "after" methods needed for the client application.

As further depicted in Figure 3, the "Barking" metaclass overrides the **sommBeforeMethod** and **sommAfterMethod** with a method that emits one bark when invoked. Thus, one can create the "BarkingDog" class, whose instances (such as "Lassie") bark twice when "disturbed" by a method invocation.

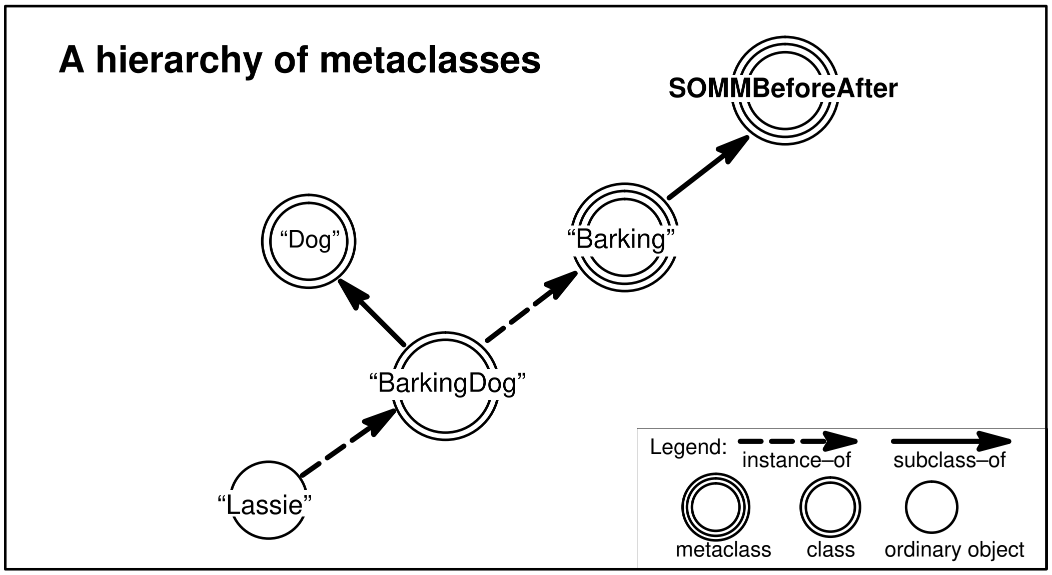

Figure 3. A hierarchy of metaclasses

The **SOMMBeforeAfter** metaclass is designed to be subclassed; a subclass (or child) of **SOMMBeforeAfter** is also a metaclass. The subclass overrides **sommBeforeMethod** or **sommAfterMethod** or both. These (redefined) methods are invoked before and after any method supported by instances of the subclass (these methods are called *primary* methods). That is, they are invoked before and after methods invoked on the ordinary objects that are instances of the class objects that are instances of the subclass of **SOMMBeforeAfter**.

The **sommBeforeMethod** returns a **boolean** value. This allows the "before" method to control whether the "after" method and the primary method get invoked. If **sommBeforeMethod** returns TRUE, normal processing occurs. If FALSE is returned, neither the primary method nor the corresponding **sommAfterMethod** is invoked. In addition, no more deeply nested before/after methods are invoked (see "Composition of before/after metaclasses" below). This facility can be used, for example, to allow a before/after metaclass to provide secure access to an object. The implication of this convention is that, if **sommBeforeMethod** is going to return FALSE, it must do any post-processing that might otherwise be done in the "after" method.

Caution: **somInit** and **somFree** are among the methods that get before/after behavior. This implies the following two obligations are imposed on the programmer of a **SOMMBeforeAfter** class. First, the implementation must guard against **sommBeforeMethod** being called before **somInit** has executed, and the object is not yet fully initialized. Second, the implementation must guard against **sommAfterMethod** being called after **somFree**, at which time the object no longer exists (see the example “C implementation for ‘Barking’ metaclass” below).

The following example shows the IDL needed to create a Barking metaclass. Just run the appropriate emitter to get an implementation binding, and then provide the appropriate “before” behavior and “after” behavior.

SOM IDL for ‘Barking’ metaclass

```
#ifndef Barking_idl
#define Barking_idl

#include <sombacls.idl>
interface  Barking : SOMMBeforeAfter
{
#ifdef __SOMIDL__
implementation
{
  //# Class Modifiers
  filestem = barking;
  callstyle = idl;

  //# Method Modifiers
  sommBeforeMethod : override;
  sommAfterMethod : override;
};
#endif /* __SOMIDL__ */
};
#endif  /* Barking_idl */
```

The next example shows an implementation of the Barking metaclass in which *no* barking occurs when **somFree** is invoked.

C implementation for ‘Barking’ metaclass

```
#define Barking_Class_Source
#include <barking.ih>

static char *somMN_somFree = ”somFree”;
static somId somId_somFree = &somMN_somFree;

SOM_Scope boolean  SOMLINK sommBeforeMethod(Barking somSelf,
                                            Environment *ev,
                                            SOMObject object,
                                            somId methodId,
                                            va_list ap)
{
    if ( !somCompareIds( methodId, somId_somFree )
       printf( ”WOOF” );
}
```

```
SOM_Scope void  SOMLINK sommAfterMethod(Barking somSelf,
                                        Environment *ev,
                                        SOMObject object,
                                        somId methodId,
                                        somId descriptor,
                                        somToken returnedvalue,
                                        va_list ap)
{
    if ( !somCompareIds( methodId, somId_somFree )
       printf( ”WOOF” );
}
```

Composition of before/after metaclasses

Consider Figure 4 in which there are two before/after metaclasses — “Barking” (as before) and “Fierce”, which has a **sommBeforeMethod** and **sommAfterMethod** that both growl (that is, both methods make a “grrrr” sound when executed). The preceding discussion demonstrated how to create a “FierceDog” or a “BarkingDog”, but has not yet addressed the question of how to compose these properties of fierce and barking. *Composability* means having the ability to easily create either a “FierceBarkingDog” that goes “grrr woof woof grrr” when it responds to a method call or a “BarkingFierceDog” that goes “woof grrr grrr woof” when it responds to a method call.

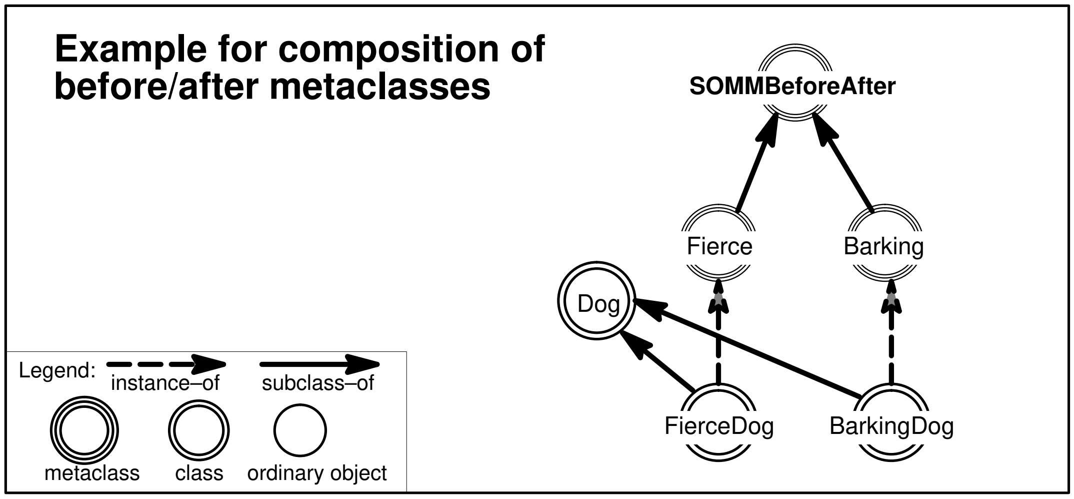

Figure 4. Example for composition of before/after metaclasses.

There are several ways to express such compositions. Figure 5 depicts SOM IDL fragments for three techniques in which composition can be indicated by a programmer. These are denoted as Technique 1, Technique 2, and Technique 3, each of which creates a “FierceBarkingDog” class, named “FB-1”, “FB-2”, and “FB-3”, respectively, as follows:

- In Technique 1, a new metaclass (“FierceBarking”) is created with both the “Fierce” and “Barking” metaclasses as parents. An instance of this new metaclass (that is, “FB-1”) is a “FierceBarkingDog” (assuming “Dog” is a parent).
- In Technique 2, a new class is created which has parents that are instances of “Fierce” and “Barking” respectively. That is, “FB-2” is a “FierceBarkingDog” also (assuming “FierceDog” and “BarkingDog” do not further specialize “Dog”).
- In Technique 3, “FB-3”, which also is a “FierceBarkingDog”, is created by declaring that its parent is a “BarkingDog” and that its explicit (syntactically declared) metaclass is “Fierce”.

Technique 1

```
interface FB-1 : Dog
{
  ...
 implementation
  {
     metaclass = FierceBarking;
       ...
   };
};
```

Technique 2

```
interface FB-2 : FierceDog,
                 BarkingDog
{
   ...
 implementation
  {
       ...
   };
};
```

Technique 3

```
interface FB-3 : BarkingDog
{
   ...
 implementation
  {
   metaclass = Fierce;
     ...
   };
};
```

Figure 5. Three techniques for composing before/after metaclasses.

Figure 6 combines the diagrams for the techniques in Figure 5 and shows the actual class relationships. Note that the explicit metaclass in the SOM IDL of “FB-1” is its derived class, “FierceBarking”. The derived metaclass of “FB-2” is also “FierceBarking”. Lastly, the derived metaclass of “FB-3” is not the metaclass explicitly specified in the SOM IDL; rather, it too is “FierceBarking.”

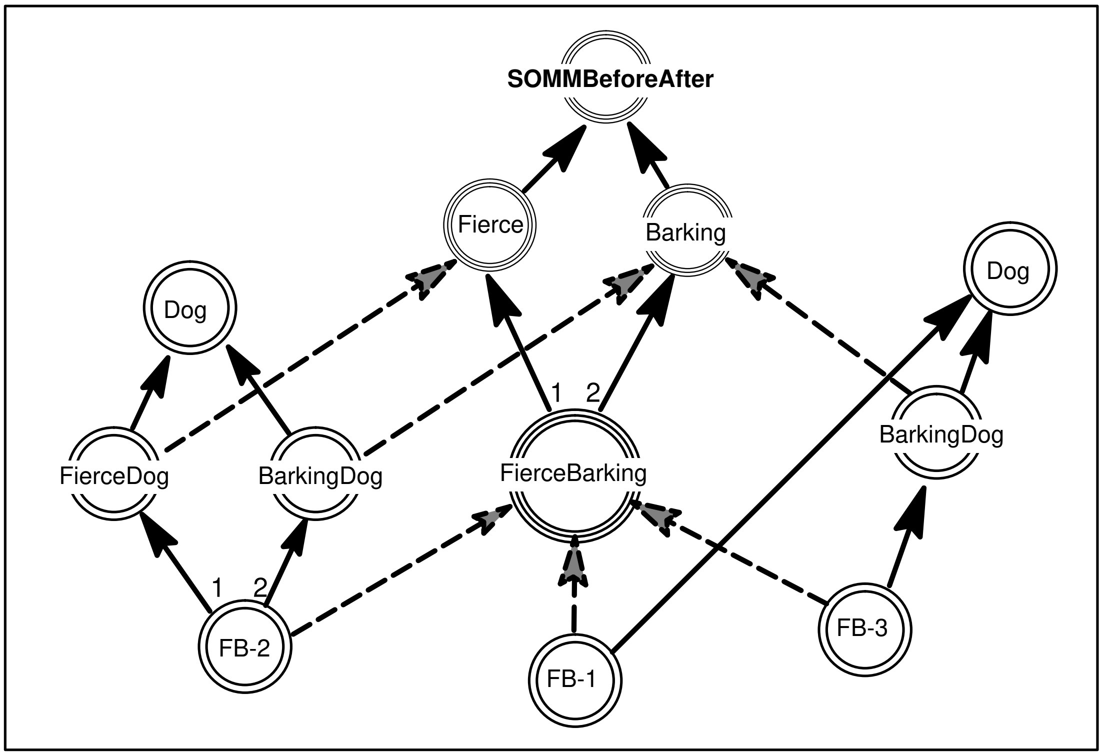

Figure 6. The combined diagram depicting the three techniques for creating a “FierceBarkingDog”.

Notes and advantages of 'before/after' usage

Notes on the dispatching of before/after methods:

- A before (after) method is invoked just once per primary method invocation.
- The dispatching of before/after methods is thread-safe.
- The dispatching of before/after methods is fast. The time overhead for dispatching a primary method is on the order of *N* times the time to invoke a before/after method as a procedure, where *N* is the total number of before/after methods to be applied.

In conclusion, consider an example that clearly demonstrates the power of the composition of before/after metaclasses. Suppose you are creating a class library that will have *n* classes. Further suppose there are *p* properties that must be included in all combinations for all classes. Potentially, the library must have $n2^p$ classes. Let us hypothesize that (fortunately) all these properties can be captured by before/after metaclasses. In this case, the size of the library is $n+p$.

The user of such a library need only produce those combinations necessary for a given application. In addition, note that there is none of the usual programming. Given the IDL for a combination of before/after metaclasses, the SOM compiler generates the implementation of the combination (in either C or C++) with no further manual intervention.

10.2 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 this 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, one might prefer the second form of creating a single-instance object to the first.

Instances of **SOMMSingleInstance** keep a count of the number of times **somNew** and **sommGetSingleInstance** are invoked. Each invocation of **somFree** decrements this count. An invocation of **somFree** does not actually free the single instance until the count reaches zero.

SOMMSingleInstance overrides **somRenew, somRenewNoInit, somRenewNoInitNoZero,** and **somRenewNoZero** so that a proxy is created in the space indicated in the **somRenew*** call. This proxy redispatches all methods to the single instance, which is always allocated in heap storage. Note that all of these methods (**somRenew***) increment the reference count; therefore, **somFree** should be called on these objects, too. In this case, **somFree** decrements the reference and frees the single instance (and, of course, takes no action with respect to the storage indicated in the original **somRenew*** call).

If a class is an instance of **SOMMSingleInstance**, all of its subclasses are also instances of **SOMMSingleInstance**. Be aware that this also means that each *subclass* is allowed to have only a single instance. (This may seem obvious. However, it is a common mistake to create a framework class that must have a single instance, while at the same time expecting users of the framework to subclass the single instance class. The result is that two single-instance objects are created: one for the framework class and one for the subclass. One technique that can mitigate this scenario is based on the use of **somSubstituteClass**. In this case, the creator of the subclass must substitute the subclass for the framework class — before the instance of the framework class is created.)

10.3 The ‘SOMMTraced’ Metaclass

SOMMTraced is a metaclass that facilitates tracing of method invocations. If class “Collie” is an instance of **SOMMTraced** (if **SOMMTraced** is the metaclass of “Collie”), any method invoked on an instance of “Collie” is traced. That is, before the method begins execution, a message prints (to standard output) giving the actual parameters. Then, after the method completes execution, a second message prints giving the returned value. This behavior is attained merely by being an instance of the **SOMMTraced** metaclass.

If the class being traced is contained in the Interface Repository, actual parameters are printed as part of the trace. If the class is not contained in the Interface Repository, an ellipsis is printed.

To be more concrete, consider Figure 7. Here, the class “Collie” is a child of “Dog” and is an instance of **SOMMTraced**. Because **SOMMTraced** is the metaclass of “Collie,” any method invoked on “Lassie” (an instance of “Collie”) is traced.

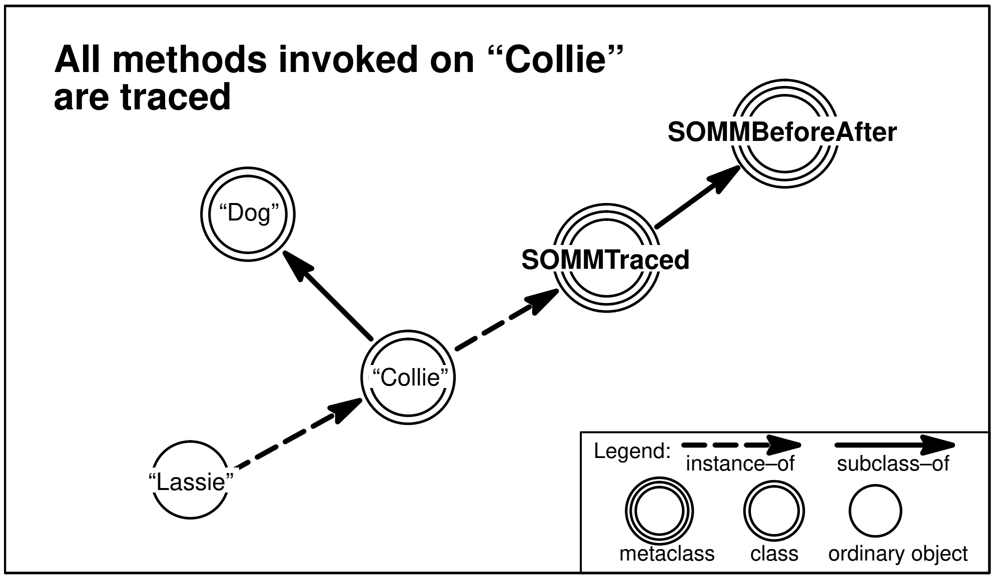

Figure 7. All methods (inherited or introduced) that are invoked on “Collie” are traced.

It is easy to use **SOMMTraced**: Just make a class an instance of **SOMMTraced** in order to get tracing.

There is one more step for using **SOMMTraced**: Nothing prints unless the environment variable SOMM_TRACED is set. If it is set to the empty string, *all* traced classes print. If SOMM_TRACED is not the empty string, it should be set to the list of names of classes that should be traced. For example, the following command turns on printing of the trace for “Collie”, but not for any other traced class:

```
export  SOMM_TRACED=Collie       (on AIX)
SET     SOMM_TRACED=Collie       (on OS/2 or Windows)
```

The example below shows the IDL needed to create a traced dog class: Just run the appropriate emitter to get an implementation binding.

SOM IDL for 'TracedDog' class

```
#include "dog.idl"
#include <somtrcls.idl>
interface TracedDog : Dog
{
#ifdef __SOMIDL__
implementation
{
  //# Class Modifiers
  filestem = trdog;
  metaclass = SOMMTraced;
};
#endif /* __SOMIDL__ */
};
```

10.4 The ‘SOMRReplicable’ Metaclass

A second testimonial to the usefulness of before/after metaclasses is the encapsulation of the Replication Framework. If you reflect on the recipe for converting a class into a replicable class (see Chapter 9, “The Replication Framework”), you will observe that the basic idea is to wrap each method between calls to the Replication Framework. This is accomplished with a combination of the metaclass **SOMRReplicable** and the class **SOMRReplicableObject**.

Figure 8 depicts the way in which the Replication Framework is encapsulated by these two new classes and how they are used to create a replicable class “ReplicableDog.” Observe that the creator of “ReplicableDog” is only obligated to make “ReplicableDog” a subclass of **SOMRReplicableObject**. The remainder of the structure is shown for the benefit of those readers who intend to create their own before/after metaclasses.

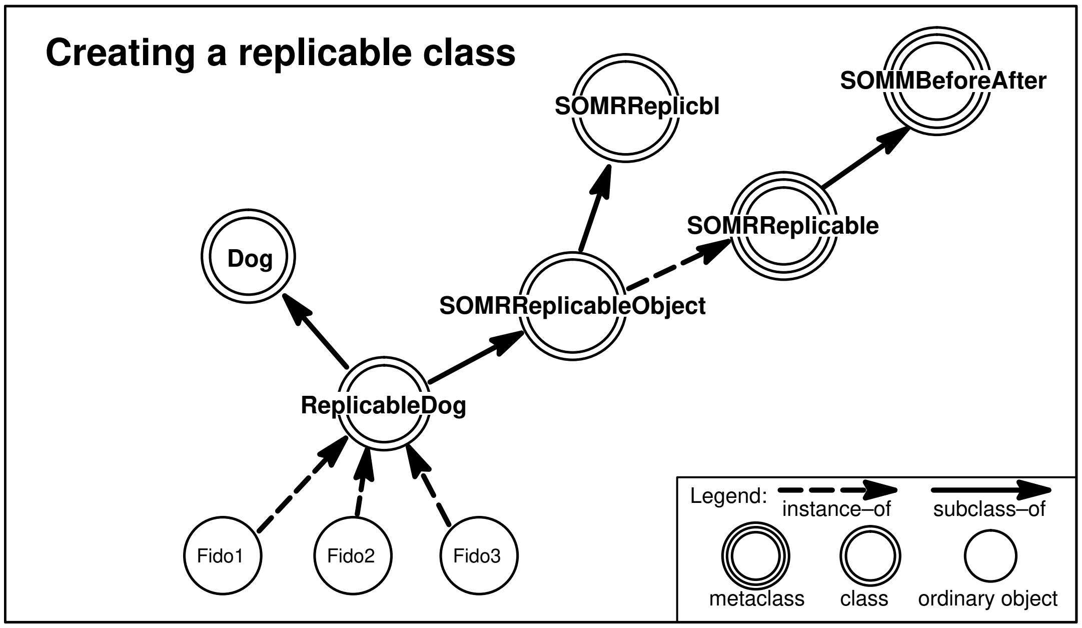

Figure 8. Creating a replicable class.

The examples that follow show the IDL and the C implementation for creating the class “ReplicableDog” of Figure 8 (the implementation of methods inherited from **SOMRReplicbl** are omitted). There are several points about these examples that should be noted:

- The class **SOMRReplicableObject** is used as a parent and the **SOMRReplicable** metaclass is not directly referenced.
- The **somrSetObjName** method must still be a invoked on an object to assign it a name and **somrRepInit** must still be invoked on a replica to indicate the type of logging and the type of access.
- When using operation logging, you might not want all methods logged and propagated. For example, a method that returns a value without modifying the replica should not be logged and propagated. To prevent this, the **SOMRReplicableObject** class has the method **somrReplicableExemptMethod** with signature:

  boolean somrReplicableExemptMethod (in somId *methodId* **);**

  The programmer of a replicable class can override **somrReplicableExemptMethod** so that it returns TRUE if *methodId* (the **somId** of the primary method) matches that of any method you wish to exempt. The **somrReplicableExemptMethod** should return FALSE otherwise. Note that all methods supported by **SOMRReplicbl** are automatically exempted (this includes the methods supported by **SOMObject**).

- The implementation still must override **somrGetState** and **somrSetState**. In fact, when value logging is employed, **somrGetState** and **somrSetState** are used to get and set the value (which is not necessarily the case when using the Replication Framework directly).
- The **sommBeforeMethod** and **sommAfterMethod** in **SOMRReplicable** metaclass extend the set of directives with which **SOMRDoDirectives** is invoked. In addition to the directives defined in Chapter 9, there are four more: MASTER_UNREACHABLE, UNAUTHORIZED, TIMEOUT, and TRYLATER. These correspond to the return codes in the environment parameter when invoking **somrLock**, **somrLockNLogOp**, **somrReleaseNPropagateOperation**, and **somrReleaseNPropagateUpdate**.
- You can still create larger atomic operations by using the **somrPin** and **somrUnPin** methods.

SOM IDL for a 'ReplicableDog' class

```
#include ”dog.idl”
#include <somrcls.idl>
interface ReplicableDog : SOMRReplicableObject, Dog
{
#ifdef __SOMIDL__
implementation
{
  //# Class Modifiers
  filestem = repdog;
  somrGetState: override;
  somrSetState: override;
  somrDoDirective: override;
  somrReplicableExemptMethod: override;

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

C implementation for a 'ReplicableDog' class

```
#define ReplicableDog_Class_Source
#include <repdog.ih>

static char *somMN_stealthyMove = ”stealthyMove”;
static somId somId_stealthyMove = &somMN_stealthyMove;

SOM_Scope void  SOMLINK somrGetState( ReplicableDog somSelf,
                                      Environment *ev,
                                      string* buf)
{
    ...
}

SOM_Scope void  SOMLINK somrSetState( ReplicableDog somSelf,
                                      Environment *ev,
                                      string buf)
{
    ...
}
```

```
SOM_Scope void  SOMLINK somrDoDirective(
                                      ReplicableDog somSelf,
                                      Environment *ev,
                                      string str)
{
    ...
}

SOM_Scope boolean  SOMLINK somrReplicableExemptMethod(
                                      ReplicableDog somSelf,
                                      Environment *ev,
                                      somId methodId)
{
    return somCompareIds( methodId, somId_stealthyMove );
}
```

10.5 Error Codes

It is possible to receive the following messages from the Metaclass Framework while an application is running.

60001	An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somNew**. That is, some other metaclass has already claimed the right to return the value for **somNew**.
60002	An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somFree**. That is, some other metaclass has already claimed this right to override **somFree**.
60004	An invocation of **somrRepInit** was made with a logging type other than '**o**' or '**v**'.
60005	The **sommBeforeMethod** or the **sommAfterMethod** was invoked on a **SOMRReplicableObject** whose logging type is other than '**o**' or '**v**'. This error cannot occur normally. The likely cause is that some method invoked on another object has overwritten this object's memory.
60006	A Before/After Metaclass must override both **sommBeforeMethod** and **sommAfterMethod**. This message indicates an attempt to create a Before/ After Metaclass where only one of the above methods is overridden.

Chapter 11. Collection Classes

Contents

Chapter 11. Collection Classes

The Collection Classes described in this chapter constitute a large group of classes and methods provided for the programmer's convenience. Collection Classes — sometimes also called Foundation Classes — are a set of classes whose purpose is to contain other objects. These classes and their related methods implement most of the common data structures encountered in programming, thus relieving the programmer of those coding tasks. The collection classes can be used in client code "as is," or they can be used as the basis for deriving new classes.

Reference documentation for these classes and their related methods is contained the *SOMobjects Developer Toolkit Collection Classes Reference Manual.*

11.1 Categories of Collection Classes

The collection classes are organized into the following categories:

• **Abstract classes**	Define the conceptual operations that are implemented by methods in other (sub)classes.
• **Main collection classes**	Represent each of the implemented data structures for collecting elements into a group.
• **Iterator classes**	Define an iterator class corresponding to each of the main collection classes, enabling clients to iterate through each of the objects in the collection.
• **Mixin classes**	Define characteristics that apply to more than one kind of collection class, such as ordering or linking. A collection class may also require certain mixin characteristics in objects that it collects. To facilitate this, a mixin class can be "mixed in" with an existing user class to derive a new "collectible" class.
• **Supporting classes**	Provides additional capabilities used internally by collection classes; is of interest primarily to those deriving new collection classes.

Each group of the foregoing classes is discussed more thoroughly in subsequent topics, with particular emphasis on the main collection classes.

IsSame vs. IsEqual comparisons

The distinction between the **IsSame** vs. **IsEqual** operation is an important concept when making comparisons. The various collection classes use one or the other of these approaches to compare contained objects. The operations are defined as follows:

1. **IsSame** is true when two objects are really the *same* object. This means "both" objects are literally the same object; that is, both parts of a comparison are testing the same instantiation.
2. **IsEqual** is true when two objects are *equivalent* objects. This would occur when two different instantiations contain the same values, or at least values which, for the sake of the comparison, can be considered the same. Stated differently, the two instantiations are isomorphic.

Class inheritance vs. element inheritance

There are two distinct aspects of inheritance that pertain to each class:

1. The inheritance of the *collection class* itself that is derived from its parent or base class.
2. The inheritance of the *elements* or objects that can be inserted into a collection class as a value.

Do not assume that these two inheritances are the same. There are times when a collection class has one parent, but the objects to be inserted into the collection class may have a totally different parent. Further, a collection class may mandate that elements (or values) meet certain inheritance requirements before the elements can be stored in that collection container.

The subsequent topics describing each collection class also discuss any pertinent inheritance restrictions of the class and its contained elements.

Object-initializer methods

Most of the collection classes provide optional initializers. These are methods with which a newly created instance can be initialized to some state other than its default. All initializer methods use the following format:

somf<*className*>Init<*optional postfix to distinguish between initializers*>

The initializers can also be used to reset certain default properties of the collection classes. For example, although the **somf_THashTable** class uses an **IsSame** approach for comparing objects, the initializer method could be used to cause an instance of **somf_THashTable** to compare using **IsEqual** instead.

Some initializer methods cannot be overridden by inheriting classes. That is, the initializer methods can be used, but they cannot be redefined.

Naming conventions

The class names for Mixin classes all begin with the prefix **somf_M.** All other collection classes have names beginning with the prefix **somf_T.**

The method names of methods defined by the collection classes all begin with the prefix **somf**, without an underscore after the prefix.

11.2 Abstract Classes

The Annotated C++ Reference Manual[1] describes an abstract class as follows: "The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations."

The concept of an *abstract base class*, which was briefly discussed earlier in this manual, is a C++ variation on the *abstract class*. An abstract base class is a class that not only describes the general concept, it also can not be instantiated. Another aspect of the abstract base class is the notion of *pure virtual function*. Any child of the parent abstract base class *must* override each pure virtual function (method) in order to use the function.

While the idea of a pure virtual function is primarily a C++ concept, it is a valid concept in SOM as well. This is especially true when defining basic behavior in a parent class that applies for all children of that class. This concept allows class implementors the flexibility to use either of two approaches:

1. Declare an interface in the parent class to a method that all children *must* override and redefine. If the method is *not* overridden, the parent class will print a corresponding message and processing will halt.
2. Declare and define an interface in the parent to a method that the children can either accept as their base definition or can override and redefine.

The **somf_TIterator** class provides an example of an abstract class which declares a method interface in the parent that all children *must* override and redefine. Specifically, the **somf_TIterator** class declares the methods **somfFirst** and **somfNext,** which all children derived from **somf_TIterator** must override. The **somf_TIterator** class is discussed in the topic "Iterator classes," as well as in the *SOMobjects Developer Toolkit Collection Classes Reference Manual*.

The Abstract Classes include the following classes:

somf_TCollection	— Represents a group of objects.
somf_TIterator	— Declares the behavior common to all iterator classes. An iterator for a particular collection class will iterate over each element contained in an object of that class.
somf_TSequence	— Declares the behavior common to all collections whose elements are ordered.
somf_TSequenceIterator	— Declares the behavior of all iterators for children of the **somf_TSequence** class. This class is also a child of the **somf_TIterator** class.

1. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual* (Addison–Wesley Publishing Company, 1992).

11.3 Main Collection Classes

The set of main collection classes contains data structure classes for the following kinds of data structures, as described in the subsequent topics:

Hash table
Dictionary
Set
Deque, queue, and stack
Linked list
Sorted sequence
Priority queue

Hash table class — somf_THashTable

A *hash table* is a table consisting of (*key, value*) pairs. The “key” provides the means for mapping into the table, and the “value” is the data element to be stored in the hash table. A hash table data structure is implemented by the **somf_THashTable** class.

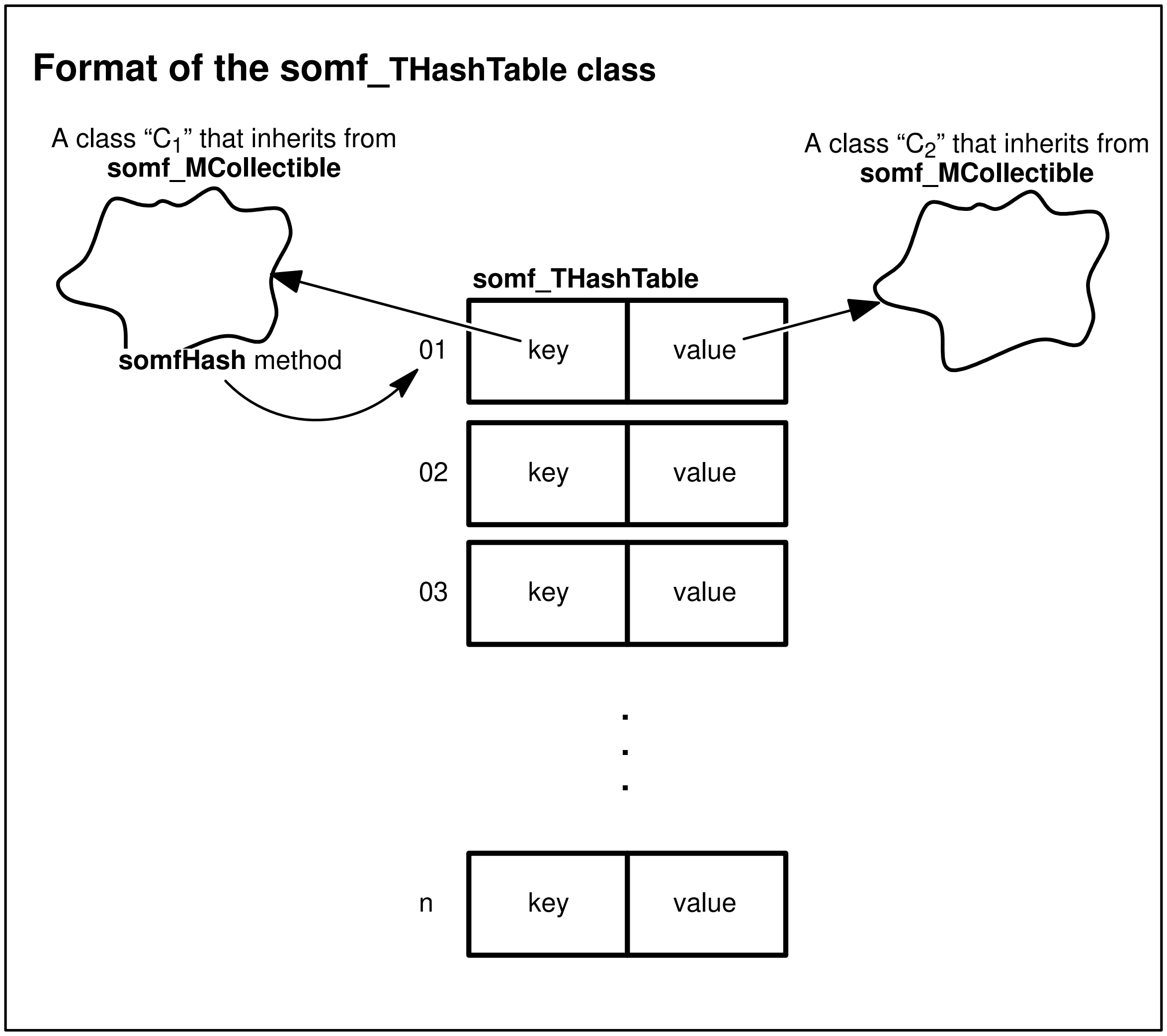
Format of the somf_THashTable class
A class “C1” that inherits from somf_MCollectible
A class “C2” that inherits from somf_MCollectible
somf_THashTable
somfHash method
01
key
value
02
key
value
03
key
value
n
key
value

Figure 1. **somf_THashTable** format

The “key” is not used directly to map into the table, because the key could be any sort of class containing a wide variety of data. Rather, the key must have a corresponding **somfHash** method (inherited from the **somf_MCollectible** class), which provides a hashing algorithm to compute the hash value (or hash probe) used to map directly into the table.

For example, if the key were a character string, the hashing algorithm could be either the number of characters in the string or the sum of the ASCII code for each letter. The programmer can choose which algorithm to use, but that algorithm must be consistent. Since the **somfHash** method will probably determine its hashing probe based on the state of a particular instance of the class, a specific state must consistently mean that **somfHash** returns the same hashing probe. If the state of the object changes, **somfHash** could return a different hashing probe, but that algorithm must then remain consistent until the state changes again. In short, once a method is used, it should consistently map to the same location thereafter, provided nothing else changes.

Both the “keys” and the “values” inserted into the **somf_THashTable** must inherit from the **somf_MCollectible** class. When the key inherits from the **somf_MCollectible** class, it overrides the **somfHash** method to provide the hashing algorithm for the table. The **somf_MCollectible** class is discussed further in the topic “Mixin classes,” as well as in the *SOMobjects Developer Toolkit Collection Classes Reference Manual,* which also provides additional information about the **somf_THashTable** class.

Objects of the **somf_THashTable** class use the **IsSame** operation to compare objects (see the earlier topic “IsSame vs. IsEqual”). This means that a single instance of the “key” can map into the hash table only once. Keys that are equal can be used without a problem, but the *same* key can only appear in the hash table once.

Note: In the event that multiple “values” are needed for the same key, you might consider storing a deque, a set, or a list as the “value.” Then, the value in the (key, value) pair would be a pointer to another collection.

Dictionary class — somf_TDictionary

A *dictionary* is an unordered data structure with (key, value) pairs. It can be thought of as the ‘cousin’ of the hash table, because they are so similar. The dictionary’s primary difference from the hash table is that the dictionary uses an **IsEqual** operation to compare objects (see the earlier topic “IsSame vs. IsEqual”). This means equal keys can only appear in the dictionary once.

The dictionary data structure is implemented by the **somf_TDictionary** class. Both the keys and values that are inserted into the **somf_TDictionary** inherit from the **somf_MCollectible** class. Just as for the hash table, when a key inherits from the **somf_MCollectible** class, it overrides the **somfHash** method to provide the hashing algorithm for the dictionary. See the descriptions of the **somf_TDictionary** and **somf_MCollectible** classes in the *SOMobjects Developer Toolkit Collection Classes Reference Manual* for more details.

Note: In the event that multiple “values” are needed for keys that are equal, you might consider storing a deque, a set, or a list as the “value.” Then, the value in the (key, value) pair would be a pointer to another collection.

Set class — somf_TSet

A *set* is an unordered collection of objects where the objects can only appear once. Note that a set does *not* contain (key,value) pairs; it only contains objects. A set is different from a deque or list, because sets involve a unique group of methods: intersections, unions, exclusive or, and differences. These are common concepts in set theory.

Set structures are implemented by the **somf_TSet** class, and the objects you can insert into them inherit from the **somf_MCollectible** class.

Deque, queue, and stack class — somf_TDeque

This class encompasses three kinds of lists:

- A *queue* is a list where the elements are inserted and removed using a first-in, first-out (FIFO) approach.
- A *stack* is a list where the elements are inserted and removed using a last-in, first-out (LIFO) approach.
- A *deque* is a double-ended queue, which permits insertion and removal at either end of the list.

All three of these data structures are implemented in the **somf_TDeque** class, with different methods processing the logically different structures. However, the **somf_TDeque** class is more than all three data structures combined, because elements can be inserted and removed from any point in the **somf_TDeque**. In addition, the **somf_TDeque** is probably the most flexible of the data structures, because an element can appear in it more than once, and the only ordering in the data structure is determined by how elements are inserted into it.

All elements inserted into the **somf_TDeque** inherit from the **somf_MCollectible** class. See the descriptions of the **somf_TDeque** and **somf_MCollectible** classes in the *SOMobjects Developer Toolkit Collection Classes Reference Manual* for more details.

Linked list class — somf_TPrimitiveLinkedList

A *linked list* is a collection where each element in the list is linked to the element in front of it and also to the element behind it. Insertion into the list is relative to the elements that are already in the list.

A linked list is implemented by the **somf_TPrimitiveLinkedList** class. This is probably one of the simplest of data structures, but consequently it is also one of the most restrictive:

- The **somf_TPrimitiveLinkedList** class is the only main collection class that does *not* inherit from **somf_MCollectible.** This cuts down on processing overhead, but it also means that an instance of **somf_TPrimitiveLinkedList** *cannot* be inserted into any other main collection class.
- Elements in the **somf_TPrimitiveLinkedList** class inherit from **somf_MLinkable**, which means the ‘linkability’ of each object is inherent in the object itself. Hence, each object can be inserted into the **somf_TPrimitiveLinkedList** only *once*, because it has only one set of unalterable forward/backward links.

For more information, see the descriptions of the classes **somf_TPrimitiveLinkedList** and **somf_MLinkable** in the *SOMobjects Developer Toolkit Collection Classes Reference Manual*.

Note: As discussed, although the **somf_TPrimitiveLinkedList** class carries no baggage from the other classes and is as compact as possible, that also means it is not very flexible. If more flexibility is needed for a linked list, you should probably consider the **somf_TDeque** class.

Sorted sequence class — somf_TSortedSequence

A *sorted sequence* is a collection where the order of its elements is determined by how those elements relate to each other. A sorted sequence data structure is implemented by the **somf_TSortedSequence** class.

Before elements can be inserted, the **somf_TSortedSequence** first must determine the relationship of the elements to each other. Therefore, elements eligible for insertion into a **somf_TSortedSequence** data structure must inherit from the **somf_MOrderableCollectible** class and must override methods **somfIsEqual**, **somfIsLessThan**, and **somfIsGreaterThan**, so that **somf_TSortedSequence** will be able to position them properly.

For more information, see the descriptions of the classes **somf_TSortedSequence** and **somf_MOrderableCollectible** in the *SOMobjects Developer Toolkit Collection Classes Reference Manual*.

Priority queue class — somf_TPriorityQueue

A *priority queue* is a special case of the sorted sequence. Its ordering is also based on how the elements relate to each other, but it is geared more toward holding larger volumes of data.

Robert Sedgewick[2] describes a priority queue as follows:

> In many applications, records with keys must be processed in order, but not necessarily in full sorted order and not necessarily all at once. Often a set of records must be collected, then the largest processed, then perhaps more records collected, then the next largest processed, and so forth. An appropriate data structure in such an environment is one that supports the operations of inserting a new element and deleting the largest element. Such a data structure, which can be contrasted with queues (delete the oldest) and stacks (delete the newest) is called a *priority queue*.

The priority queue is implemented by the **somf_TPriorityQueue** class. The unique purpose of the **somf_TPriorityQueue** is to provide a high-performance sorted sequence, to accommodate a large volume of data. A drawback is that the **somf_TPriorityQueue** class defines slightly fewer methods than does the **somf_TSortedSequence** class.

Before elements can be inserted into it, the **somf_TPriorityQueue** must first determine the relationship of those elements to each other. Thus, to be eligible for insertion into the **somf_TPriorityQueue,** elements must inherit from the **somf_MOrderableCollectible** class and must override the methods **somfIsEqual**, **somfIsLessThan**, and **somfIsGreaterThan**, so that the **somf_TPriorityQueue** will be able to position them correctly.

See the descriptions of the **somf_TPriorityQueue** and **somf_MOrderableCollectible** classes in the *SOMobjects Developer Toolkit Collection Classes Reference Manual* for more details.

Choosing the best class

If you are unsure which main collection class you should use, the following selection chart may prove helpful.

2. Robert Sedgewick, *Algorithms in C++* (Addison–Wesley Publishing Company, 1992), p. 145.

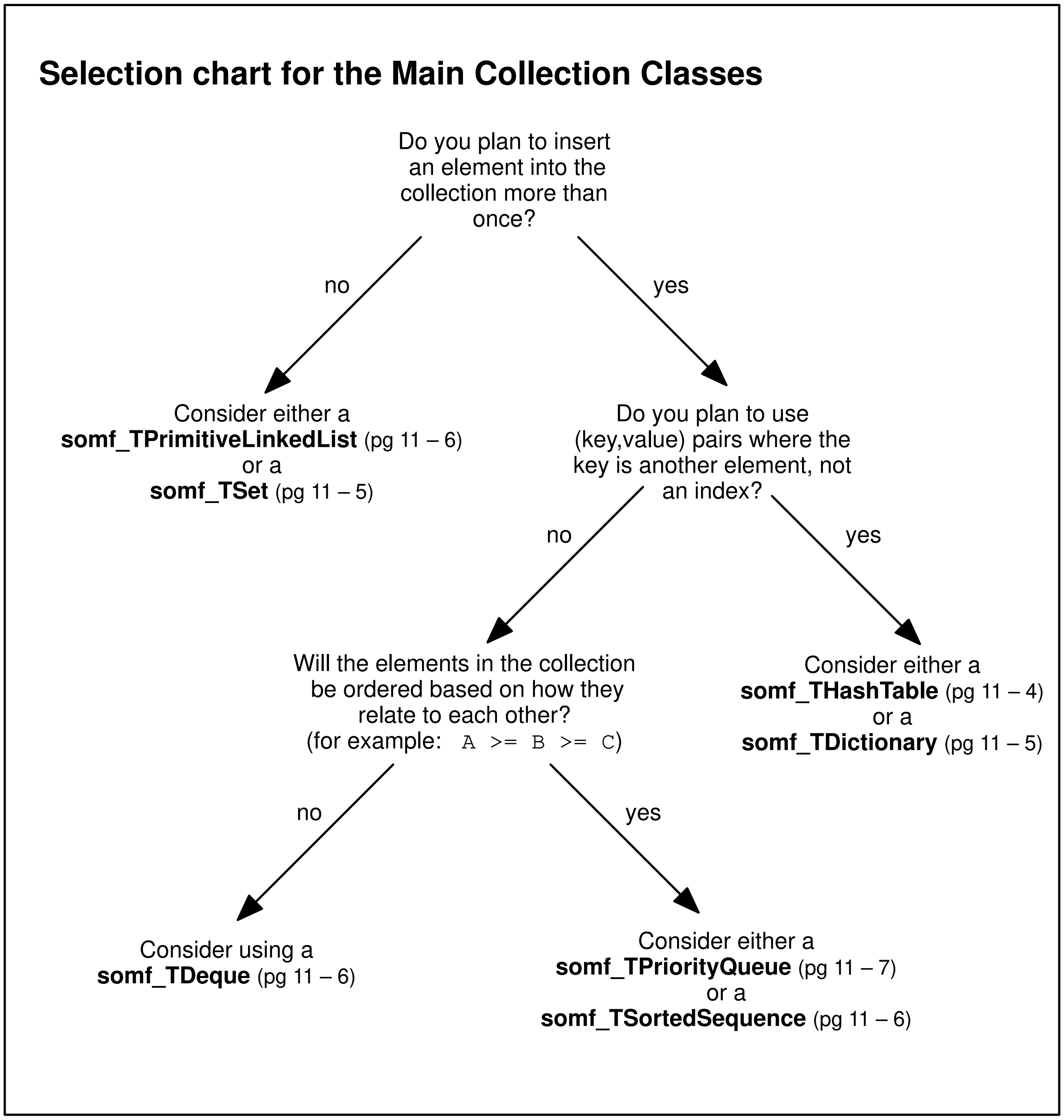
Selection chart for the Main Collection Classes
Do you plan to insert an element into the collection more than once?
no
yes
Consider either a somf_TPrimitiveLinkedList (pg 11 – 6) or a somf_TSet (pg 11 – 5)
Do you plan to use (key,value) pairs where the key is another element, not an index?
no
yes
Will the elements in the collection be ordered based on how they relate to each other? (for example: A >= B >= C)
Consider either a somf_THashTable (pg 11 – 4) or a somf_TDictionary (pg 11 – 5)
no
yes
Consider using a somf_TDeque (pg 11 – 6)
Consider either a somf_TPriorityQueue (pg 11 – 7) or a somf_TSortedSequence (pg 11 – 6)

Figure 2. Collection class selection chart

11.4 Iterator Classes

Each of the main collection classes discussed previously has a corresponding iterator class defined for it. An iterator for a particular object will iterate over each contained object in the collection. To illustrate, the following example uses an iterator of class **somf_TSetIterator** to iterate over each element in an object of the **somf_TSet** class.

```
somf_TSet set;
somf_TSetIterator itr;
somf_MCollectible obj;
Environment *ev;

set = somf_TSetNew();
ev = somGetGlobalEnvironment();

/* A bunch of stuff happens to set */

itr = somf_TSet_somfCreateIterator(set,ev);
obj = _somfFirst(itr,ev);
while (obj != SOMF_NIL)
{
        /* do something to obj */
        obj = _somfNext(itr,ev);
}
```

The **somfFirst** method is used to get the first element in the collection, and the **somfNext** method is used thereafter to get each "next" element. Using an iterator allows you to sequentially look at each element in the collection and do some appropriate processing on it.

Notice that the iterator was initialized using the method **somfCreateIterator**. All classes that inherit from **somf_TCollection** must provide a **somfCreateIterator** method. This shows one of the two ways to initialize an iterator; the other way is to use the constructor-initializer associated with the iterator. For example, `itr` could have been declared using:

```
somf_TSetIterator itr;
itr = somf__TSetIteratorNew();
somf_TSetIterator_somfTSetIteratorInit(itr, ev, set);
```

Note: Some people may wonder why the iterator logic was not included in the main collection classes, rather than being in a separate class. One reason was done so the user can create multiple iterators for a single instance of a collection class. If the methods were part of the main collection classes, each instance would be limited to the one iterator that came with it.

If a collection changes while the iterator is in use, the iterator becomes invalid and will issue a notice that it cannot continue to the next element. So, for example, if a client program calls the collection's **somfAdd** method after starting to iterate through the collection, the iterator will not allow processing to continue. The iterator will have to be reset, and the easiest way to do that is to call the iterator's **somfFirst** method and start over.

If a collection is ordered, the iterator returns its elements in the correct order. If the collection is unordered or partially ordered (like **somf_TPriorityQueue**), the iterator returns its elements in some random order. This is discussed in more detail in the *SOMobjects Developer Toolkit Collection Classes Reference Manual.*

11.5 Mixin Classes

A Mixin class is a class designed "to be mixed in together with other classes to produce new subclasses."[3] Mixin classes do not necessarily describe stand-alone characteristics — just characteristics that may be common to more than one kind of class. For example, classes describing a "car" or a "dress" might both inherit from a Mixin class describing "red."

Another characteristic of mixin classes is that they inherit *only* from other mixin classes (not including **SOMObject**). A mixin class can *not* inherit from some other base class without a **somf_M** prefix.

For any object to be eligible for insertion into one of the main collection classes, that object *must* inherit from a Mixin class (see the table below). This is necessary because the mixin class declares certain behavior that the main collection class requires in the object in order to process it. For example, the **somf_MCollectible** mixin class declares the **somfIsEqual** method that is needed to compare objects in almost every collection.

To utilize a collection class for storing their own objects, programmers must first define a class whose instances will contain the required mixin characteristics. By using multiple inheritance, a class can inherit from zero or more mixin classes, as well as from its logical parent (if it has a logical parent). For example, if you want to store objects in a sorted sequence structure of class **somf_TSortedSequence**, those objects must be instances of a class defined to inherit characteristics from the **somf_MOrderableCollectible** mixin class, such as:

```
interface MySortSeqData : MyData, somf_MOrderableCollectible
```

The *SOMobjects Developer Toolkit Collection Classes Reference Manual* includes a "Description" topic for each class, and it is vital that the user read this before using each main collection class. This topic describes what methods of a mixin class *must* be overridden in any objects that will be inserted into the collection class, if the collection class is to work properly.

There are three important mixin classes used by the main collection classes:

somf_MCollectible	— Defines the generic methods needed by objects inserted into *any* of the collections classes. It provides the profile for the methods **somfIsEqual**, **somfIsSame**, and **somfHash**.
somf_MLinkable	— Defines the general characteristics of objects that contain links.
somf_MOrderableCollectible	— Defines the general characteristics of objects that are ordered.

The following table maps each main collection class to the mixin class from which an object must inherit in order for that object to be eligible for insertion into the corresponding main collection class:

Main Collection Class	**Mixin Class from which an inserted object must inherit**
somf_TDeque	**somf_MCollectible**
somf_TDictionary	**somf_MCollectible**
somf_THashTable	**somf_MCollectible**
somf_TPrimitiveLinkedList	**somf_MLinkable**
somf_TPriorityQueue	**somf_MOrderableCollectible**
somf_TSet	**somf_MCollectible**
somf_TSortedSequence	**somf_MOrderableCollectible**

3. Grady Booch, *Object Oriented Design with Applications* (Redwood City, California: The Benjamin/Cummings Publishing Company, 1991), pg. 58

11.6 Supporting Classes

Many of the main collection classes use supporting classes. The **somf_TSortedSequence** class, for example, uses the supporting class **somf_TSortedSequenceNode** to define the behavior of a single node in a sorted sequence collection.

Included are the following supporting classes:

somf_TAssoc	— Is used to hold a pair of objects.
somf_TDequeLinkable	— Inherits from **somf_MLinkable** and provides a generic version of **somf_MLinkable** containing a long value. The **somf_TDequeLinkable** class is used by **somf_TDeque**.
somf_TSortedSequenceNode	— Represents a node in a tree containing elements of the **somf_MOrderableCollectible** class. It contains a key (the **somf_MOrderableCollectible**) and a link to a left child and a right child.
somf_TCollectibleLong	— Provides a generic **somf_MCollectible** class containing a long value.

somf_TDequeLinkable and **somf_TSortedSequenceNode** will probably not be of particular interest unless you plan to derive a new collection class. **somf_TAssoc** may only be of interest if you are working with **somf_THashTable** or **somf_TDictionary**, since these two classes store key, value pairs. **somf_TCollectibleLong** will be of interest if you need a generic **somf_MCollectible** containing a long. **somf_TCollectibleLong** is not used by any of the other Collection Classes.

11.7 Inheritance Hierarchy of the Collection Classes

The inheritance hierarchy for the collection classes is depicted in the following chart. Note that this diagram does *not* illustrate all of the classes, only those which have some position in the inheritance hierarchy of the set.

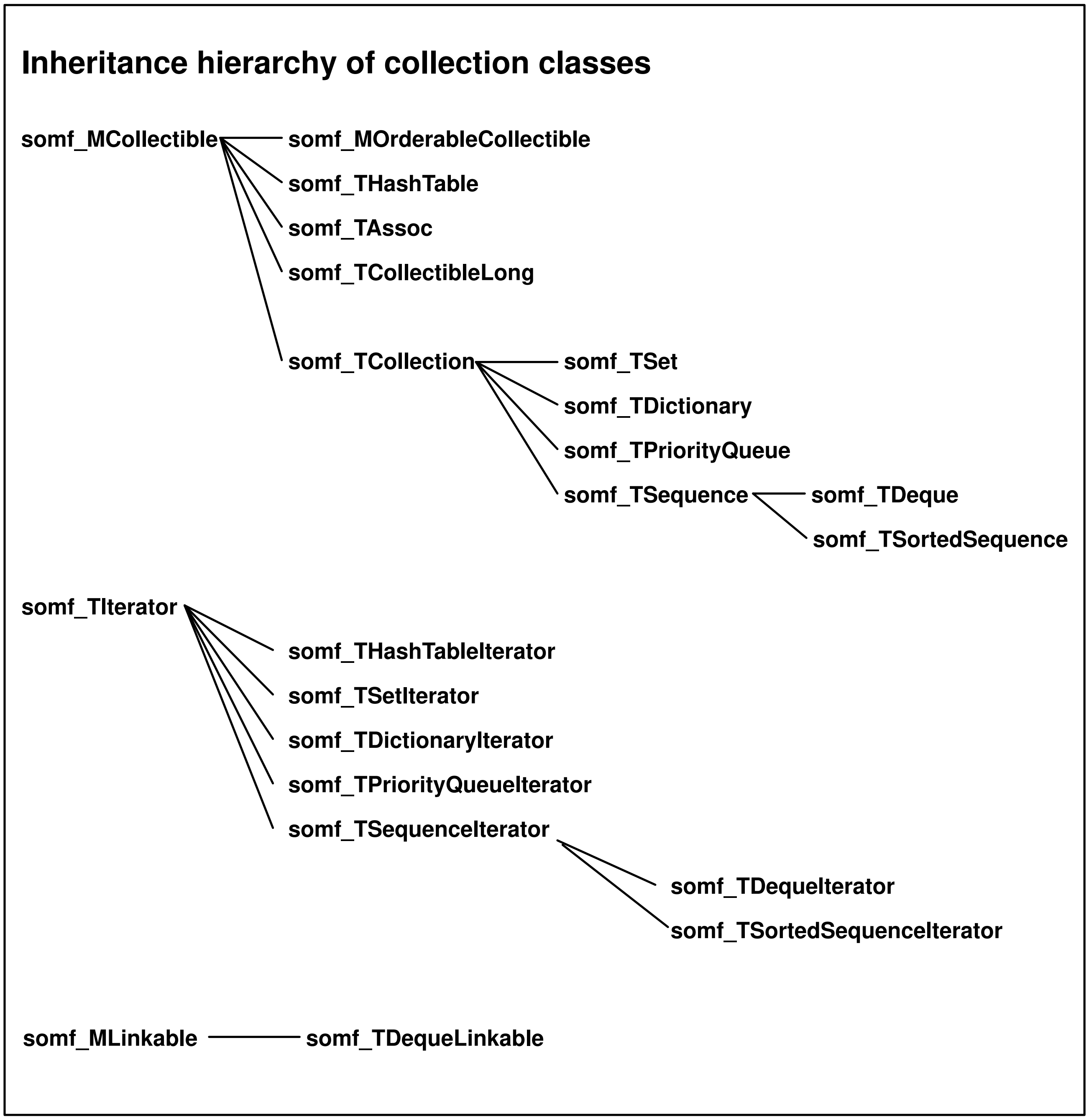

Figure 3. Collection classes inheritance hierarchy

11.8 Utility Collection Classes by Category

Following is the entire list of utility collection classes. Each class is described previously in this chapter, as well as in the *SOMobjects Developer Toolkit Collection Classes Reference Manual.*

Abstract Classes

somf_TCollection
somf_TIterator
somf_TSequence
somf_TSequenceIterator

Main Collection Classes

somf_TDeque
somf_TDictionary
somf_THashTable
somf_TPrimitiveLinkedList
somf_TPriorityQueue
somf_TSet
somf_TSortedSequence

Iterator Classes

somf_TDequeIterator
somf_TDictionaryIterator
somf_THashTableIterator
somf_TPrimitiveLinkedListIterator
somf_TPriorityQueueIterator
somf_TSetIterator
somf_TSortedSequenceIterator

Mixin Classes

somf_MCollectible
somf_MLinkable
somf_MOrderableCollectible

Supporting Classes

somf_TAssoc
somf_TCollectibleLong
somf_TDequeLinkable
somf_TSortedSequenceNode

Chapter 12. The Event Management Framework

Contents

Chapter 12. 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 Replication Framework and by DSOM to wait on their respective events of interest. The Event Management Framework must also be used by any interactive application that contains DSOM or replicated objects.

12.1 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.

Note: Under Windows, a single event processing loop must necessarily incorporate the program's message processing loop. See "Processing Events" below for a description of how this is accomplished by 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 and Windows, only TCP/IP sockets are supported.

  Note: On OS/2 and Windows, the Sockets classes for NetBIOS (**NBSockets**) and Novell IPX/SPX (**IPXSockets**) are primarily intended for use by DSOM and the Replication Framework, not for general application programming.
- **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 SOMLINK 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.  */
/* On Windows, the SOMLINK keyword is NOT included if the
 * application is intended to support multiple instances. */
```

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 5, Section 5.3 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,
          (EMRegProc *) 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

You 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”);
      }
  }
     /* On OS/2, ”system” linkage is also required.  */
     /* On Windows, callbacks do not use the SOMLINK keyword if
      * the application is intended to support multiple instances. */
```

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.

For Windows:
The **_someProcessEvents** method incorporates a standard window-message-processing loop into its event processing loop. In order to allow timer events to operate, messages are retrieved using **PeekMessage** rather than **GetMessage**. If you do not wish to use timer events, or if you want to perform other processing in your message loop (such as, translating accelerators), then you should use **_someProcessEvent**, as follows:

```
while (GetMessage(&msg, NULL, NULL, NULL))
{
   if (!TranslateAccelerator(hwndMain, haccel, &msg))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   _someProcessEvent(some_gEMan, Ev, EMProcessTimerEvent    |
                                     EMProcessSinkEvent     |
                                     EMProcessClientEvent   |
                                     EMProcessWorkProcEvent );
    ...
}
```

Interactive applications

Note: This topic does *not* apply to SOMobjects For Windows, as user input is processed in the message loop that is incorporated into EMan.

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.)

12.2 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. See the example program in Chapter 9, “The Replication Framework” (section 9.7).

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 **SOMEEMRegisterData** 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.

12.3 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** (and **TCPIPSockets32** for OS/2), **NBSockets**, and **IPXSockets**.