Object Lifetime Manager

A Complementary Pattern for Controlling Object Creation and Destruction

David L. Levine and Christopher D. Gill
{levine,cdgill}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer
Engineering Department
University of California, Irvine, USA*

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2000. Abridged versions of the paper appeared at the Pattern Languages of Programming Conference, Allerton Park, Illinois, USA, 15 – 18 August 1999 and in the C++ Report magazine, January, 2000.

1 Introduction

Creational patterns such as Singleton and Factory Method [1] address object construction and initialization, but do not consider object destruction. In some applications, however, object destruction is as important as object construction. The *Object Lifetime Manager* pattern addresses issues associated with object destruction. Object Lifetime Manager is also an example of a *complementary pattern*, which completes or extends other patterns. In particular, the Object Lifetime Manager pattern completes creational patterns by considering the entire lifetime of objects.

This paper is organized as follows: Section 2 describes the Object Lifetime Manager pattern in detail using the Siemens format [2]. and Section 3 presents concluding remarks.

2 The Object Lifetime Manager Pattern

2.1 Intent

The *Object Lifetime Manager* pattern can be used to govern the entire lifetime of objects, from creating them prior to their first use to ensuring they are destroyed properly at program termination. In addition, this pattern can be used to replace static object creation/destruction with dynamic object preallocation/deallocation that occurs automatically during application initialization/termination.

2.2 Example

Singleton [1] is a common creational pattern that provides a global point of access to a unique class instance and defers creation of the instance until it is first accessed. If a singleton is not needed during the lifetime of a program, it will not be created. The Singleton pattern does not address the issue of when its instance is destroyed, however, which is problematic for certain applications and operating systems.

To illustrate why it is important to address destruction semantics, consider the following logging component that provides a client programming API to a distributed logging service [3]. Applications use the logging component as a front-end to the distributed logging service to report errors and generate debugging traces.

```
class Logger
{
public:
  // Global access point to Logger singleton.
  static Logger *instance (void) {
    if (instance_ == 0)
      instance_ = new Logger;
    return instance_;
  }

  // Write some information to the log.
  int log (const char *format, ...);

protected:
  // Default constructor (protected to
  // ensure Singleton pattern usage).
  Logger (void);

  static Logger *instance_;
  // Contained Logger singleton instance.

  // . . . other resources that are
  //       held by the singleton . . .
};
```

*This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and Nortel.

```
// Initialize the instance pointer.
Logger *Logger::instance_ = 0;
```

The `Logger` constructor, which is omitted for brevity, allocates various OS endsystem resources, such as socket handles, shared memory segments, and/or system-wide semaphores, that are used to implement the logging service client API.

To reduce the size and improve the readability of its logging records, an application may choose to log certain data, such as timing statistics, in batch mode rather than individually. For instance, the following statistics class batches timing data for individual identifiers:

```
class Stats
{
public:
  // Global access point to the
  // statistics singleton.
  static Stats *instance (void) {
    if (instance_ == 0)
      instance_ = new Stats;
    return instance_;
  }

  // Record a timing data point.
  int record (int id,
              const timeval &tv);

  // Report recorded statistics
  // to the log.
  void report (int id) {
    Logger::instance ()->
      log ("Avg timing %d: "
           "%ld sec %ld usec\n",
           id,
           average_i (id).tv_sec,
           average_i (id).tv_usec);
  }

protected:
  // Default constructor.
  Stats (void);

  // Internal accessor for an average.
  const timeval &average_i (void);

  // Contained Stats singleton instance.
  static Stats *instance_;

  // . . . other resources that are
  //       held by the instance . . .
};

// Initialize the instance pointer.
Stats *Stats::instance_ = 0;
```

After recording various statistics, a program calls the `Stats::report` method, which uses the `Logger` singleton to report average timing statistics for an identifier.

Both the `Logger` and `Stats` classes provide distinct services to the application: the `Logger` class provides general logging capabilities, whereas the `Stats` class provides specialized batching and logging of time statistics. These classes are designed using the Singleton pattern, so that a single instance of each is used in an application process.

The following example illustrates how an application might use the `Logger` and `Stats` singletons.

```
int main (int argc, char *argv[])
{
  // Interval timestamps.
  timeval start_tv, stop_tv;

  // Logger, Stats singletons
  // do not yet exist.

  // Logger and Stats singletons created
  // during the first iteration.
  for (int i = 0; i < argc; ++i) {
    ::gettimeofday (&start_tv);
    // do some work between timestamps . . .
    ::gettimeofday (&stop_tv);
    // then record the stats . . .
    timeval delta_tv;
    delta_tv.sec = stop_tv.sec - start_tv.sec;
    delta_tv.usec = stop_tv.usec - start_tv.usec;
    Stats::instance ()->record (i, delta_tv);

    // . . . and log some output.
    Logger::instance ()->
      log ("Arg %d [%s]\n", i, argv[i]);
    Stats::instance()->report (i);
  }

  // Logger and Stats singletons are not
  // cleaned up when main returns.
  return 0;
}
```

Note that the `Logger` and `Stats` singletons are not constructed or destroyed explicitly by the application, *i.e.*, their lifetime management is decoupled from the application logic. It is common practice to not destroy singletons at program exit [4].

Several drawbacks arise, however, from the fact that the Singleton pattern only addresses the *creation* of singleton instances and does not deal with their destruction. In particular, when the `main` program above terminates, neither the `Logger` nor the `Stats` singletons are cleaned up. At best, this can lead to false reports of leaked memory. At worst, important system resources may not be released and destroyed properly.

For instance, problems can arise if the `Logger` and/or `Stats` singletons hold OS resources, such as system-scope semaphores, I/O buffers, or other allocated OS resources. Failure to clean up these resources gracefully during program shutdown can cause deadlocks and other synchronization hazards. To alleviate this problem, each singleton's destructor should be called before the program exits.

One way of implementing the Singleton pattern that attempts to ensure singleton destruction is to employ the Scoped Locking C++ idiom [5] that declares a static instance of the class at file scope [4]. For example, the following `Singleton Destroyer` template provides a destructor that deletes the singleton.

```
template <class T>
Singleton_Destroyer
{
public:
  Singleton_Destroyer (void): t_ (0) {}
  void register (T *) { t_ = t; }
  ~Singleton_Destroyer (void) { delete t_; }
private:
  T *t_; // Holds the singleton instance.
};
```

To use this class, all that's necessary is to modify the `Logger` and `Stats` classes by defining a static instance of the `Singleton Destroyer`, such as the following example for `Logger`:

```
static Singleton_Destroyer<Logger>
  logger_destroyer;

// Global access point to the
// Logger singleton.
static Logger *instance (void) {
  if (instance_ == 0) {
    instance_ = new Logger;
    // Register the singleton so it will be
    // destroyed when the destructor of
    // logger_destroyer is run.
    logger_destroyer.register (instance_);
  }
  return instance_;
}

// . . . similar changes to Stats class . . .
```

Note how `logger_destroyer` class holds the singleton and deletes it when the program exits. A similar `Singleton_Destroyer` could be used by the `Stats` singleton, as well.

Unfortunately, there are several problems with explicitly instantiating static `Singleton_Destroyer` instances. In C++, for example, each `Singleton_Destroyer` could be defined in a different compilation unit. In this case, there is no guaranteed order in which their destructors will be called, which can lead to undefined program behavior. In particular, if singletons in different compilation units share resources, such as socket handles, shared memory segments, and/or system-wide semaphores, the program may fail to exit cleanly. The undefined order of singleton destruction in C++ makes it hard to ensure these resources are released by the OS before (1) the last singleton using the resource is completely destroyed, but not before (2) a singleton that is still alive uses the resource(s).

In summary, the key forces that are not resolved in these examples above are: (1) resources allocated by a singleton must ultimately be released when a program exits, (2) unconstrained creation and destruction order of static instances can result in serious program errors, and (3) shielding software developers from responsibility for details of object lifetime management can make systems less error-prone.

2.3 Context

An application or system where full control over the lifetime of the objects it creates is necessary for correct operation.

2.4 Problem

Many applications do not handle the entire lifetime of their objects properly. In particular, applications that use creational patterns, such as Singleton, often fail to address object destruction. Similarly, applications that use static objects to provide destruction often suffer from inconsistent initialization and termination behavior. These problems are outlined below.

Problems with singleton destruction: Singleton instances may be created dynamically.[1] A dynamically allocated singleton instance becomes a resource leak if it is not destroyed, however. Often, singleton leaks are ignored because (1) they aren't significant in many applications and (2) on most multi-user general-purpose operating systems, such as UNIX or Windows NT, they are cleaned up when a process terminates.

Unfortunately, resource leaks can be troublesome in the following contexts:

- **When graceful shutdown is required [4]:** Singletons may be responsible for system resources, such as system-wide locks, open network connections, and shared memory segments. Explicit destruction of these singletons may be desirable to ensure these resources are destroyed at a well-defined point during program termination. For instance, if the `Logger` class in Section 2.2 requires system-wide locks or shared memory, then it should release these resources after they are no longer needed.

- **When singletons maintain references to other singletons:** Explicitly managing the order of destruction of singletons may be necessary to avoid problems due to dangling references during program termination. For example, if the `Stats` class in the example above uses the `Logger` instance in its `report` method, this method could be invoked during the `Stats` instance destruction, which renders the behavior of the program undefined. Likewise, to support useful behaviors, such as logging previously unreported values during program shutdown, the termination ordering of these singletons must be controlled.

[1] Singleton is used as an example in much of this pattern description because (1) it is a popular creational pattern and (2) it highlights challenging object destruction issues nicely. However, Object Lifetime Manager can complement other creational patterns, such as Factory Method, and does not assume that its managed objects are singletons or of homogeneous types.

• **When checking for memory leaks:** Memory leak detection tools, *e.g.*, NuMega BoundsCheck, ParaSoft Insure++, and Rational Purify, are useful for languages such as C and C++ that require explicit allocation and deallocation of dynamic memory. These tools identify singleton instances as leaked memory, reports of which obscure important memory leaks.

For instance, if many identifiers are used to record `Stats` data in our running example, it may appear that a sizable amount of memory is leaking during program operation. In large-scale applications, these "leaks" can result in numerous erroneous warnings, thereby obscuring real memory leaks and hindering system debugging.

• **Dynamic memory allocation may be from a global pool:** Some real-time operating systems, such as VxWorks [6] and pSOS [7], have only a single, global heap for all applications. Therefore, application tasks must release dynamically allocated memory upon task termination; otherwise, it cannot be reallocated to other applications until the OS is rebooted. Failure to explicitly release memory that was allocated dynamically by the `Logger` and `Stats` singletons in our running example represents real resource leaks on such platforms.

Problems with static object lifetime: Some objects must be created prior to any use. In C++, such instances traditionally have been created as *static objects*, which are intended to be constructed prior to invocation of the main program entry point and destroyed at program termination. However, there are several important drawbacks to static objects:

• **Unspecified order of construction/destruction.** C++ only specifies the order of construction/destruction of static objects *within* a compilation unit (file); the construction order matches the declaration order and destruction is the reverse order [8]. However, there is no constraint specified on the order of construction/destruction *between* static objects in different files. Therefore, construction/destruction ordering is implementation-dependent. For example, the versions of the `Logger` and `Stats` classes that use the `Singleton Destroyer` in Section 2.2 illustrate problems that arise from the undefined order of destruction of the `Stats` and `Logger` singleton instances.

It is hard to write portable C++ code that uses static objects possessing initialization dependencies. Often, it is simpler to avoid using static objects altogether, rather than trying to analyze for, and protect against, such dependencies. This approach is particularly appropriate for reusable components and frameworks [9], which should avoid unnecessary constraints on how they are used and/or initialized.

Explicit singleton management is necessary for correct program operation on some platforms because they destroy singletons prematurely. For instance, the garbage collector in older Java Development Kits (JDKs) may destroy an object when there are no longer any references to it, even if the object was intended to be a singleton [10]. Though this deficiency has been fixed in later JDKs, the Object Lifetime Manager could solve it, under application control, by maintaining singleton references.

Another problem in Java applications is the sharing of namespaces, which allow sharing (intended or otherwise) between singletons in separate applets [11]. Again, the Object Lifetime Manager can be used to register singleton instances. Applets would then access their singletons from this registry.

• **Poor support by embedded systems.** Embedded systems have historically used C. Therefore, they do not always provide seamless support for OO programming language features. For instance, the construction/destruction of static objects in C++ is one such feature that often complicates embedded systems programming. The embedded OS may have support for explicit invocation of static constructor/destructor calls, but this is not optimal from a programmer's perspective.

Some embedded operating systems do not support the notion of a *program* that has a unique entry point. For example, VxWorks supports multiple *tasks*, which are similar to threads because they all share one address space. However, there is no designated *main* task for each application. Therefore, these embedded systems platforms can be configured to call static constructors/destructors at module (object file) load/unload time, respectively. On such platforms, it is not otherwise necessary to unload and load between repeated executions. To properly destroy and construct static objects, however, the static object destructors/constructors must either be called manually, or the module unloaded and loaded again, which hinders repeated testing.

In addition, placement of data in read-only memory (ROM) complicates the use of static objects [12]. The data must be placed in ROM prior to run-time; however, static constructors are called at run-time. Therefore, embedded systems sometimes do not support calls of static constructors and destructors. Moreover, if they are supported it may be under explicit application control, instead of by implicit arrangement of the compiler and run-time system.

• **Static objects extend application startup time.** Static objects may be constructed at application startup time, prior to invocation of the main entry point. If these objects are not used during a specific execution, then application construction (and destruction) times are needlessly extended. One way to eliminate this waste is to replace each such static object with one that is allocated on demand, *e.g.*, by using the Singleton pattern.

Replacement of a static object with a singleton also can be used to delay construction until the first use of the object.

Again, this reduces startup time. Some real-time applications may find it advantageous to construct the singletons at a specific time, after the main entry point has been entered, but before the objects are needed.

One or more of these drawbacks of static objects typically provides sufficient motivation for removing them from a program. Often, it is better not use them in the first place, but to apply the following solution instead.

2.5 Solution

Define an *Object Lifetime Manager*, which is a singleton that contains a collection of *Preallocated Objects* and *Managed Objects*. The Object Lifetime Manager is responsible for constructing and destroying the Preallocated Objects at program initialization and termination, respectively. It is further responsible for ensuring all of its Managed Objects are destroyed properly at program termination.

2.6 Applicability

Use Object Lifetime Manager when:

Singletons and other dynamically created objects must be removed without application intervention at program termination: Singleton and other creational patterns do not typically address the question of when the objects they create should be removed, or who should remove them. In contrast, Object Lifetime Manager provides a convenient, global object that deletes dynamically created objects. Creational pattern objects can then register with the Object Lifetime Manager for deletion, which usually occurs at program termination.

Static objects must be removed from the application: As described in Section 2.4, static objects can be troublesome, especially in some languages and on some platforms. Object Lifetime Manager provides a mechanism to replace static objects with Preallocated Objects. Preallocated Objects are dynamically allocated before the application uses them, and deallocated at program termination.

The platform does not support static object construction/destruction: Some embedded platforms, such as VxWorks and pSOS, do not always construct static objects at program initialization and destroy them at program termination.[2] In general, it is best to remove all static objects, *e.g.*, to support repeated testing of a program. Another situation where static objects can cause difficulty is when they are placed in ROM. Objects in ROM cannot be initialized at run-time, because they cannot be modified at all.

[2] On VxWorks and pSOS, static objects can be constructed when the module is loaded and destroyed when it is unloaded. After loading, an entry point can be called more than once before unloading. Therefore, a *program* can be run more than once after constructing static objects, without ever destroying them. Conversely, static object constructors and destructors can be invoked explicitly.

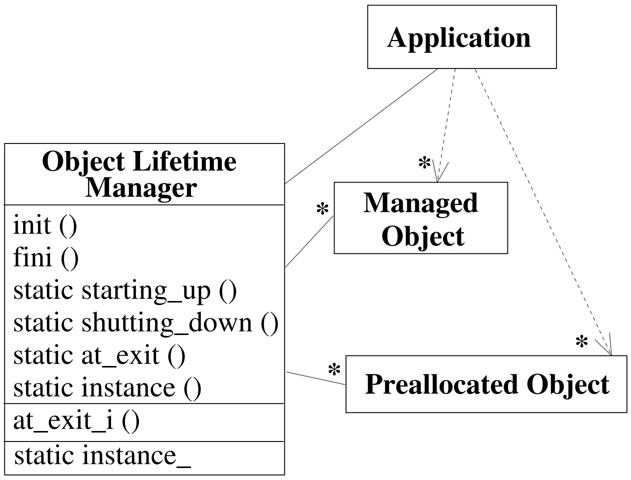
The underlying platform does not provide a notion of a main program, although the application needs it: The lack of support for static object construction/destruction on some platforms stems from their lack of support for the notion of a program, as discussed in Section 2.4. The Object Lifetime Manager pattern can be used to emulate programs by partitioning this address space. The scope of each Object Lifetime Manager delineates a program, from an application's perspective.

Destruction order must be specified by the application: Dynamically created objects can be *registered* with the Object Lifetime Manager for destruction. The Object Lifetime Manager can be implemented to destroy objects in any desired order.

The application requires explicit singleton management: As described in Section 2.4, singletons may be destroyed prematurely, for example, on earlier Java platforms. The Object Lifetime Manager delays singleton destruction until program termination.

2.7 Structure and Participants

The structure and participants of the Object Lifetime Manager pattern are shown using UML in the following figure and described below.



Object Lifetime Manager: Each `Object Lifetime Manager` is a singleton that contains collections of `Managed Objects` and `Preallocated Objects`.

Managed Objects: Any object may be *registered* with an `Object Lifetime Manager`, which is responsible for destroying the object. Object destruction occurs when the

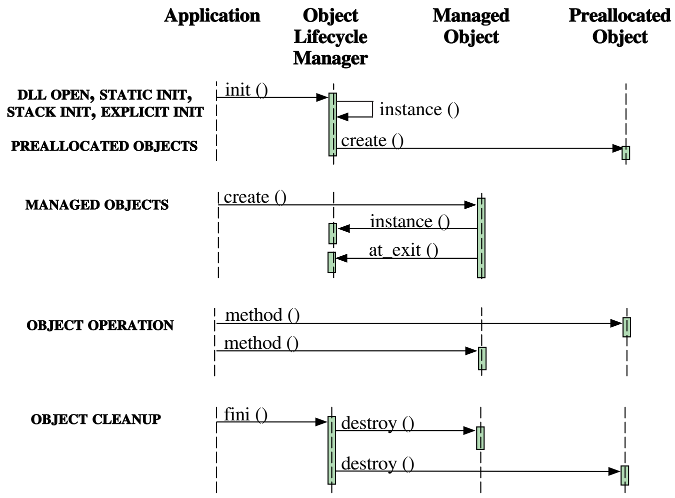
Object Lifetime Manager itself is destroyed, typically at program termination.

Preallocated Objects: An object may be hard-coded for construction and destruction by an Object Lifetime Manager. Preallocated Objects have the same lifetime as the Object Lifetime Manager, *i.e.*, the lifetime of the process that executes the program.

Application: The Application initializes and destroys its Object Lifetime Managers, either implicitly or explicitly. In addition, the Application registers its Managed Objects with an Object Lifetime Manager, which may also contain Preallocated Objects.

2.8 Dynamics

The dynamic collaborations among participants in the Object Lifetime Manager pattern are shown in the following figure. The diagram depicts four separate activities:



1. Object Lifetime Manager creation and initialization, which in turn creates the Preallocated Objects;
2. Managed Object creation by the Application, and registration with the Object Lifetime Manager;
3. Use of Preallocated and Managed Objects by the Application; and
4. Destruction of the Object Lifetime Manager, which includes destruction of all the Managed and Preallocated Objects it controls.

Within each activity, time increases down the vertical axis.

2.9 Implementation

The Object Lifetime Manager pattern can be implemented using the steps presented below. This implementation is based on the Object Manager provided in the ACE framework [9], which motivates many interesting issues discussed in this section. Some of the steps discussed below are language-specific because ACE is written in C++. Appendix A illustrates even more concretely how the Object Lifetime Manager pattern has been implemented in ACE.

1. Define the Object Lifetime Manager component. This component provides applications with an interface with which to register objects whose lifetime must be managed to ensure proper destruction upon program termination. In addition, this component defines a repository that ensures proper destruction of its managed objects. The Object Lifetime Manager is a container for the Preallocated Objects and for the Managed Objects that are registered to be destroyed at program termination.

The following substeps can be used to implement the Object Lifetime Manager.

- **Define an interface for registering Managed Objects:** One way to register Managed Objects with the Object Lifetime Manager would be to use the C-library atexit function to invoke the termination functions at program exit. However, not all platforms support atexit. Furthermore, atexit implementations usually have a limit of 32 registered termination functions. Therefore, the Object Lifetime Manager should support the following two techniques for registering Managed Objects with a container that holds these objects and cleans them up automatically at program exit:

1. *Define a cleanup function interface* – The Object Lifetime Manager allows applications to register arbitrary types of objects. When a program is shut down, the Object Lifetime Manager cleans up these objects automatically.

   The following C++ class illustrates a specialized CLEANUP_FUNC used in ACE to register an object or array for cleanup:

```
typedef void (*CLEANUP_FUNC)(void *object,
                             void *param);

class Object_Lifetime_Manager
{
public:
  // . . .
  static int at_exit (void *object,
                      CLEANUP_FUNC cleanup_hook,
                      void *param);
  // . . .
};
```

The static `at_exit` method registers an object or array of objects for cleanup at process termination. The `cleanup_hook` argument points to a global function or static method that is called at cleanup time to destroy the object or array. At destruction time, the `Object Lifetime Manager` passes the `object` and `param` arguments to the `cleanup_hook` function. The `param` argument contains any additional information needed by the `cleanup_hook` function, such as the number of objects in the array.

2. *Define a cleanup base class interface* – This interface allows applications to register for destruction with the `Object Lifetime Manager` any object whose class derives from a `Cleanup` base class. The `Cleanup` base class should have a virtual destructor and a `cleanup` method that simply calls `delete this`, which in turn invokes all derived class destructors. The following code fragment illustrates how this class is implemented in ACE:

```
class Cleanup
{
public:
  // . . .

  // Destructor.
  virtual ~Cleanup (void);

  // By default, simply deletes this.
  virtual void cleanup (void *param = 0);
};
```

The following code fragment illustrates the `Object Lifetime Manager` interface used to register objects derived from the `Cleanup` base class in ACE:

```
class Object_Lifetime_Manager
{
public:
  // . . .
  static int at_exit (Cleanup *object,
                      void *param = 0);
  // . . .
};
```

This static `at_exit` method registers a `Cleanup` object for cleanup at process termination. At destruction time, the `Object Lifetime Manager` calls the `Cleanup` object's `cleanup` method, passing in the `param` argument. The `param` argument contains any additional information needed by the `cleanup` method.

• **Define a singleton adapter:** Although it is possible to explicitly code singletons to use the `Object Lifetime Manager` methods defined above, this approach is tedious and error-prone. Therefore, it is useful to define a `Singleton` adapter class template that encapsulates the details of creating singleton objects and registering them with the `Object Lifetime Manager`. In addition, the `Singleton` adapter can ensure the thread-safe Double-Checked Locking Optimization pattern [13] is used to construct and access an instance of the type-specific `Singleton`.

The following code fragment illustrates how a singleton adapter is implemented in ACE:

```
template <class TYPE>
class Singleton : public Cleanup
{
public:
  // Global access point to the
  // wrapped singleton.
  static TYPE *instance (void) {
    // Details of Double Checked
    // Locking Optimization omitted . . .
    if (singleton_ == 0) {
      singleton_ = new Singleton<TYPE>;

      // Register with the Object Lifetime
      // Manager to control destruction.
      Object_Lifetime_Manager::
        at_exit (singleton_);
    }
    return &singleton_->instance_;
  }

protected:
  // Default constructor.
  Singleton (void);

  // Contained instance.
  TYPE instance_;

  // Instance of the singleton adapter.
  static Singleton<TYPE> *singleton_;
};
```

The `Singleton` class template is derived from the `Cleanup` class. This allows the `Singleton` instance to register itself with the `Object Lifetime Manager`. The `Object Lifetime Manager` then assumes responsibility for dynamically deallocating the `Singleton` instance and with it the adapted `TYPE` instance.

• **Define an interface for registering Preallocated Objects:** Preallocated Objects must always be created before the application's main processing begins. For instance, in some applications, synchronization locks must be created before their first use in order to avoid race conditions. Thus, these objects must be hard-coded into each `Object Lifetime Manager` class. Encapsulating their creation within the `Object Lifetime Manager`'s own initialization phase ensures this will occur, without adding complexity to application code.

The `Object Lifetime Manager` should be able to preallocate objects or arrays. The `Object Lifetime Manager` can either perform these preallocations statically in global data or dynamically on the heap. An efficient implementation is to store each Preallocated Object in an array. Certain languages, such as C++, do not support arrays of heterogeneous objects, however. Therefore, in these languages pointers must be stored instead of the objects themselves. The actual objects are allocated dynamically by the `Object Lifetime Manager` when it is instantiated, and destroyed by the `Object Lifetime Manager` destructor.

The following substeps should be used to implement `Preallocated Objects`:

1. *Limit exposure* – To minimize the exposure of header files, identify the Preallocated Objects by macros or enumerated literals, *e.g.*,

   ```
   enum Preallocated_Object_ID
     {
       ACE_FILECACHE_LOCK,
       ACE_STATIC_OBJECT_LOCK,
       ACE_LOG_MSG_INSTANCE_LOCK,
       ACE_DUMP_LOCK,
       ACE_SIG_HANDLER_LOCK,
       ACE_SINGLETON_NULL_LOCK,
       ACE_SINGLETON_RECURSIVE_THREAD_LOCK,
       ACE_THREAD_EXIT_LOCK,
     };
   ```

   The use of an enumerated type is appropriate when all preallocated objects are known *a priority*.

2. *Use cleanup adapters* – The `Cleanup Adapter` class template is derived from the `Cleanup` base class and wraps types not derived from the `Cleanup` base class so that they can be managed by the `Object Lifetime Manager`. Use template functions or macros for allocation/deallocation, *e.g.*,

   ```
   #define PREALLOCATE_OBJECT(TYPE, ID) {\
     Cleanup_Adapter<TYPE> *obj_p;\
     obj_p = new Cleanup_Adapter<TYPE>;\
     preallocated_object[ID] = obj_p;\
   }

   #define DELETE_PREALLOCATED_OBJECT(TYPE, ID)\
     cleanup_destroyer (\
       static_cast<Cleanup_Adapter<TYPE> *>\
         (preallocated_object[ID]), 0);\
     preallocated_object[ID] = 0;
   ```

   The `Cleanup Adapter` adapts any object to use the simpler `Object Lifetime Manager` registration interface, as discussed in Appendix A. Similarly, `Cleanup Destroyer` uses the `Cleanup Adapter` to destroy the object.

   An analogous array, enum, and macro pair can be supplied for preallocated arrays, if necessary.

3. *Define an accessor interface to the Preallocated Objects* – Applications need a convenient and typesafe interface to access preallocated objects. Because the `Object Lifetime Manager` supports preallocated objects of different types, it is necessary to provide a separate class template adapter with a member function that takes the id under which the object was preallocated in the `Object Lifetime Manager`, and returns a correctly typed pointer to the preallocated object.

   The following code fragment illustrates how this interface is provided via a class template adapter in ACE:

   ```
   template <class TYPE>
   class Preallocated_Object_Interface
   {
   public:
     static TYPE *
     get_preallocated_object
       (Object_Lifetime_Manager::
          Preallocated_Object_ID id)
     {
       // Cast the return type of the object
       // pointer based on the type of the
       // function template parameter.
       return
         &(static_cast<Cleanup_Adapter<TYPE> *>
           (Object_Lifetime_Manager::
             preallocated_object[id]))->object ();
     }
     // . . . other methods omitted.
   };
   ```

• **Determine the destruction order of registered objects:** As noted in Section 2.6, the `Object Lifetime Manager` can be implemented to destroy registered objects in any desired order. For example, priority levels could be assigned and destruction could proceed in decreasing order of priority. An interface could be provided for objects to set and change their destruction priority.

We have found that destruction in reverse order of registration has been a sufficient policy for ACE applications. An application can, in effect, specify destruction order by controlling the order in which it registers objects with the `Object Lifetime Manager`.

• **Define a termination function interface:** Lifetime management functionality has been discussed so far in terms of destruction of objects at program termination. However, the `Object Lifetime Manager` can provide a more general capability – the ability to call a function at program termination – using the same internal implementation mechanism.

For example, to ensure proper cleanup of open Win32 WinSock sockets at program termination, the `WSACleanup` function must be called. A variation of the Scoped Locking C++ idiom [5] could be applied, by creating a special wrapper facade [14] class whose constructor calls the API initialization functions, and whose destructor calls the API cleanup functions. The application can then register an instance of this class with the `Object Lifetime Manager` so that the object is destroyed and API termination methods are called during `Object Lifetime Manager` termination.

However, this design may be overkill for many applications. Moreover, it can be error-prone because the application must ensure the class is used only as a singleton so that the API functions are only called once. Finally, this design may increase the burden on the application to manage object destruction order, so that the singleton is not destroyed before another object that uses the API in its destructor.

Instead, the API termination functions can be called as part of the termination method of the `Object Lifetime Manager` itself. This latter approach is illustrated by the `socket_fini` call in the `Object_Lifetime_Manager::fini` method in Appendix A. The `socket_fini` function provides a platform-specific implementation that calls WSACleanup on Win32 platforms, and does nothing on other platforms.

• **Migrate common interfaces and implementation details into a base class:** Factoring common internal details into an `Object Lifetime Manager Base` class can make the `Object Lifetime Manager` implementation simpler and more robust. Defining an `Object Lifetime Manager Base` class also supports the creation of multiple `Object Lifetime Managers`, each of a separate type. To simplify our discussion, we only touch on the use of multiple `Object Lifetime Managers` briefly. They do not add consequences to the pattern, but are useful for partitioning libraries and applications.

2. Determine how to manage the lifetime of the Object Lifetime Manager itself. The `Object Lifetime Manager` is responsible for initializing other global and static objects in a program. However, that begs the important bootstrapping question of how this singleton initializes and destroys itself. The following are the alternatives for initializing the `Object Lifetime Manager` singleton instance:

• **Static initialization:** If an application has no static objects with constraints on their order of construction or destruction, it's possible to create the `Object Lifetime Manager` as a static object. For example, ACE's `Object Lifetime Manager` can be created as a static object. The ACE library has no other static objects that have constraints on order of construction or destruction.

• **Stack initialization:** When there is a main program thread with well defined points of program entry and termination, creating the `Object Lifetime Manager` on the stack of the main program thread can simplify program logic for creating and destroying the `Object Lifetime Manager`. This approach to initializing the `Object Lifetime Manager` assumes that there is one unique main thread per program. This thread defines the program: *i.e.*, it is *running* if, and only if, the main thread is alive. This approach has a compelling advantage: the `Object Lifetime Manager` instance is automatically destroyed via any path out of `main`.[3]

Stack initialization is implemented transparently in ACE via a preprocessor macro named `main`. The macro renames the `main` program entry point to another, configurable name, such as `main_i`. It provides a `main` function that creates the `Object Lifetime Manager` instance as a local object (on the run-time stack) and then calls the renamed application entry point.

There are two drawbacks to the Stack initialization approach:

1. `main (int, char *[])` must be declared with arguments, even if they're not used. All of ACE uses this convention, so only applications must be concerned with it.
2. If there are any static objects depending on those that are destroyed by the `Object Lifetime Manager`, their destructors might attempt to access the destroyed objects. Therefore, the application developer is responsible for ensuring that no static objects depend on those destroyed by the `Object Lifetime Manager`.

• **Explicit initialization:** In this approach, create the `Object Lifetime Manager` explicitly under application control. The `Object Lifetime Manager init` and `fini` methods allow the application to create and destroy the Object Lifetime Manager when desired. This option alleviates complications that arise when using dynamic link libraries (DLL)s.

• **Dynamic link library initialization:** In this approach, create and destroy the Object Lifetime Manager when its DLL is loaded and unloaded, respectively. Most dynamic library facilities include the ability to call (1) an initialization function when the library is loaded and (2) a termination function when the library is unloaded. However, see Section 2.13 for a discussion of the consequences of managing singletons from an Object Lifetime Manager in a different DLLs.

2.10 Example Resolved

The discussion in Section 2.9 demonstrates how the Object Lifetime Manager pattern can be used to resolve key design forces related to managing object lifetimes. In particular, the unresolved forces described in Section 2.2 can be satisfied by applying the Object Lifetime Manager pattern.

The following example shows how the Object Lifetime Manager pattern can be applied to the original `Logger` and

[3] On platforms such as VxWorks and pSOS that have no designated `main` function, the main thread can be simulated by instantiating the `Object Lifetime Manager` on the stack of one thread, which is denoted by convention as *the* `main` thread.

Stats examples from Section 2.2. Using the Singleton adapter template described in Section 2.9 greatly simplifies managed object implementations by encapsulating key implementation details, such as registration with the Object Lifetime Manager. For instance, the original Stats class can be replaced by a managed Stats class, as follows.

```
class Stats
{
public:
  friend class Singleton<Stats>;

  // Destructor: frees resources.
  ~Stats (void);

  // Record a timing data point.
  int record (int id,
              const timeval &tv);

  // Report recorded statistics
  // to the log.
  void report (int id) {
    Singleton<Logger>::instance ()->
      log ("Avg timing %d: "
           "%ld sec %ld usec\n",
           id,
           average_i (id).tv_sec,
           average_i (id).tv_usec);
  }

protected:
  // Default constructor.
  Stats (void)
  {
    // Ensure the Logger instance
    // is registered first, and will be
    // cleaned up after, the Stats
    // instance.
    Singleton<Logger>::instance ();
  }

  // Internal accessor for an average.
  const timeval &average_i (void);

  // . . . other resources that are
  //       held by the instance . . .
};
```

Notice that the singleton aspects have been factored out of the original Stats class and are now provided by the Singleton adapter template. Similar modifications can be made to the original Logger class so that it uses the Singleton adapter template.

Finally, the following example shows how an application might use the Logger and Stats classes.

```
int main (int argc, char *argv[])
{
  // Interval timestamps.
  timeval start_tv, stop_tv;

  // Logger and Stats singletons
  // do not yet exist.

  // Logger and then Stats singletons
  // are created and registered on the first
  // iteration.
  for (int i = 0; i < argc; ++i) {
    ::gettimeofday (&start_tv);
    // do some work between timestamps ...
    ::gettimeofday (&stop_tv);
    // then record the stats ...
    timeval delta_tv;
    delta_tv.sec = stop_tv.sec - start_tv.sec;
    delta_tv.usec = stop_tv.usec - start_tv.usec;
    Singleton<Stats>::instance ()->
      record (i, delta_tv);

    // . . . and log some output.
    Singleton<Logger>::instance ()->
      log ("Arg %d [%s]\n", i, argv[i]);
    Singleton<Stats>::instance()->report (i);
  }

  // Logger and Stats singletons are
  // cleaned up by Object Lifetime Manager
  // upon program exit.
  return 0;
}
```

The following key forces are resolved in this example: (1) ensuring resources allocated by an instance are subsequently released, (2) managing the order of creation and destruction of singletons, and (3) providing a framework that encapsulates these details within a well-defined interface.

2.11 Known Uses

Object Lifetime Manager is used in the Adaptive Communication Environment (ACE) [9] to ensure destruction of singletons at program termination and to replace static objects with dynamically allocated, managed objects. ACE is used on many different OS platforms, some of which do not support static object construction/destruction for every program invocation. ACE can be configured to not contain any objects whose initialization is necessary prior to program invocation.

Gabrilovich [15] augmented the Singleton pattern to permit applications to specify destruction order. A local static auto_ptr[4] is responsible for the destruction of each singleton instance. Destruction of singleton instances proceeds by application-defined phases; an application may optionally register its singleton instances for destruction in a specific phase.

An interesting example of a "small" Object Lifetime Manager is the *strong pointer* [16, 17].[5] A strong pointer manages just one object; it destroys the object when its scope is exited, either normally or via an exception. There can be many strong pointers in a program, behaving as Function-as-Owner (or Block-as-Owner) (see Section 2.12. Moreover, the strong

[4]The auto_ptr is a local static object in the singleton instance accessor method.

[5]A C++ auto_ptr is an implementation of a strong pointer.

pointers themselves have transient lifetimes, *i.e.*, that of their enclosing blocks.

In contrast, there is typically just one Object Lifetime Manager per program (or per large-scale component). And, Object Lifetime Managers live for the duration of the program invocation. This reflects the specific intent of the Object Lifetime Manager to destroy objects at program termination, but not sooner. Such objects may be used after the current block or function has been exited, and destruction/creation cycles are not possible or desired.

2.12 See Also

The Object Lifetime Manager pattern is related to the Manager [18] pattern. In both patterns, a client application uses a collection of objects, relying upon a manager to encapsulate the details of how the objects themselves are managed. This separation of concerns makes the application more robust, because management aspects that are potentially error-prone are hidden behind a type-safe interface. Using these managers also makes the application more extensible, because certain details of the managed objects can be varied independent of the manager implementation. For example, a manager for a certain class can be used to manage objects of classes derived from that base class.

The Object Lifetime Manager pattern differs from the Manager pattern in the types of the managed objects. Whereas the Manager pattern requires that the managed objects have a common base type, the Object Lifetime Manager pattern allows objects of unrelated types to be managed. The Manager pattern relies on inheritance for variations in the manager and managed object classes. In contrast, the Object Lifetime Manager relies on object composition and type parameterization to achieve greater decoupling of the manager from the managed objects.

The Object Lifetime Manager pattern further differs from the Manager pattern in the details of the object management services it provides to applications. The Manager pattern provides both key-based and query-based search services, *e.g.*, ISBN lookup or finding a list of books by an author [18], while the Object Lifetime Manager pattern supports a simpler, more generic key-based lookup service. The Manager pattern supports fine-grained control of object creation and destruction by the application, while the Object Lifetime Manager pattern only supports destruction of its registered objects at the end of the program.

The application can *register* a pre-existing object with the Object Lifetime Manager, which then assumes responsibility for the remaining lifetime of the managed object. The Manager pattern only allows on-demand creation of objects, so that the lifetime of objects is managed in an all-or-none manner.

These distinctions in object types and service details reflect the different intents of the two patterns. Fundamentally, the Manager pattern focuses on the search structure aspects of object management, whereas the Object Lifetime Manager pattern emphasizes the lifetime aspects instead. The Manager pattern should be used to provide tailored object lifetime services for a well defined collection of related objects. The Object Lifetime Manager pattern should be used to *complement* creational patterns, providing more generic object lifetime services for a wider collection of possibly unrelated objects.

Object Lifetime Manager complements creational patterns, such as Singleton, by managing object instance destruction. Singleton addresses only part of the object lifetime because it just manages instance creation. However, destruction is usually not an important issue with Singleton because it does not retain *ownership* of created objects [4]. Ownership conveys the responsibility for managing the object, including its destruction. Singleton is the prototypical example of a creational pattern that does not explicitly transfer ownership, yet does not explicitly arrange for object destruction. Object Lifetime Manager complements Singleton by managing the destruction portion of the object lifetime.

Object Lifetime Manager can complement other creational patterns, such as Abstract Factory and Factory Method. Implementations of these patterns could register dynamically allocated objects for deletion at program termination. Alternatively (or additionally), they could provide interfaces for object destruction, corresponding to those for object creation.

Cargill presented a taxonomy of the dynamic C++ object lifetime [19]. The Localized Ownership pattern language includes patterns, such as Creator-as-Owner, Sequence-of-Owners, and Shared Ownership, which primarily address object ownership. Ownership conveys the responsibility for destruction.

Creator-as-Owner is further subdivided into Function-as-Owner, Object-as-Owner, and Class-as-Owner. The Singleton destruction capability of Object Lifetime Manager may be viewed as new category of Creator-as-Owner: Program-as-Owner. It is distinct from Function-as-Owner, because static objects outlive the program entry point (`main`). Object Lifetime Manager's Preallocated Objects similarly can be viewed logically, at least, as outliving the main program function.

When Singleton is used on multi-threaded platforms, a mutex should be used to serialize access to the instance pointer. Double-Checked Locking [13] greatly reduces the use of this mutex by only requiring it prior to creation of the singleton. However, the mutex is still required, and it must be initialized.[6] Object Lifetime Manager solves the chicken-and-egg

[6]POSIX 1003.1c [20] mutexes can be initialized without calling a static constructor. However, they are not available on all platforms.

problem of initialization of the mutex by preallocating one for each singleton, or group of singletons.

Object Lifetime Manager uses Adapter for type-safe storage of objects of any class. By using inline functions, the `Managed Object` Adapter should have no size/performance overhead. We confirmed this with the GreenHills 1.8.9 compiler for VxWorks on Intel targets. However, some compilers do not inline template member functions. Fortunately, the size overhead of `Managed Object` is very small, *i.e.*, we measured 40 to 48 bytes with g++ on Linux and LynxOS. The ACE `Cleanup Adapter` template class has slightly higher size overhead, about 160 bytes per instantiation.

2.13 Consequences

The **benefits** of using Object Lifetime Manager include:

Destruction of Singletons and other Managed Objects at program termination: The Object Lifetime Manager pattern allows a program to shut down cleanly, releasing memory for `Managed Objects`, along with the resources they hold at program termination. All heap-allocated memory can be released by the application. This supports repeated testing on platforms where heap allocations outlive the program.[7] It also eliminates the memory-in-use warnings reported for singletons by memory access checkers at program termination.

Specification of destruction order: The order of destruction of objects can be specified. The order specification mechanism can be as simple or as complex as desired. As noted in Section 2.9, simple mechanisms are generally sufficient in practice.

Removal of static objects from libraries and applications: Static objects can be replaced by `Preallocated Objects`. This prevents applications from relying on the order in which static objects are constructed/destructed. Moreover, it allows code to target embedded systems, which sometimes have little or no support for constructing/destroying static objects.

However, the following **liabilities** must be considered when using the Object Lifetime Manager pattern:

Lifetime of the manager itself: The application must ensure that is respects the lifetime of the Object Lifetime Manager, and does not attempt to use its services outside that lifetime. For example, the application must not attempt to access `Preallocated Objects` prior to the complete initialization of the `Object Lifetime Manager`. Similarly, the application must not destroy the `Object Lifetime Manager` prior to the application's last use of a `Managed` or `Preallocated Object`. Finally, the implementation of the `Object Lifetime Manager` is simplified if it can assume that it will be initialized by only one thread. This precludes the need for a static lock to guard its initialization.

Use with shared libraries: On platforms that support loading and unloading shared libraries at run-time, the application must be *very* careful of platform-specific issues that impact the lifetime of the `Object Lifetime Manager` itself. For example, on Windows NT, the `Object Lifetime Manager` should be initialized by the application or by a DLL that contains it. This avoids a potential deadlock situation due to serialization within the OS when it loads DLLs.

A related issue arises with singletons that are created in DLLs, but managed by an `Object Lifetime Manager` in the main application code. It the DLL is unloaded before program termination, the `Object Lifetime Manager` would try to destroy it using code that is no longer linked into the application. For this reason, we have added an unmanaged Singleton class to ACE. An unmanaged Singleton is of the conventional design, *i.e.*, it does not provide implicit destruction. ACE uses a managed Singleton by default because we found the need for unmanaged Singletons to be very unusual.

3 Concluding Remarks

Many creational patterns specifically address only object *creation*. They do not consider when or how to *destroy* objects that are no longer needed. The Object Lifetime Manager pattern provides mechanisms for object destruction at program termination. Thus, it complements many creational patterns by covering the entire object lifetime.

The Singleton pattern provides a notable example where coordinated object lifetime management is important. In particular, deletion at program termination ensures that programs have no memory leaks of singleton objects. Moreover, applications that employ the Object Lifetime Manager pattern do not require use of static object constructors and destructors, which is important for embedded systems. In addition, the Object Lifetime Manager pattern supports replacement of static objects with dynamically `Preallocated Objects`, which is useful on embedded platforms and with OO languages, such as C++.

One of the Object Lifetime Manager pattern's more interesting aspects is that it addresses weaknesses of another pattern, at least in some contexts. Our initial motivation was to remedy these weaknesses by registering and deleting singletons at program termination. The utility, applicability, novelty, and complexity of the ACE `Object Lifetime Manager`

[7] On some operating systems, notably some real-time operating systems, there is no concept of a *program*. There are tasks, *i.e.*, threads, but no one task has any special, main identity. Thus, there is no cleanup of dynamically allocated memory, open files, *etc.*, at task termination.

class seemed to be on par with those of the ACE `Singleton` adapter template class, so we felt that it deserved consideration as a pattern. Because it can address just part of the object lifetime, however, we consider Object Lifetime Manager to be a *complementary* pattern.

Another interesting question was: "How do we categorize Object Lifetime Manager?" It was not (originally) a Creational pattern, because it handled only object destruction, not creation. Again, it seemed appropriate to refer to the Object Lifetime Manager pattern as complementing the Singleton pattern.

In addition, we realized that `Object Lifetime Manager` had another use that was related to destroying singletons. Static objects create problems similar to those of singletons, *i.e.*, destruction, especially on operating systems that have no notion of a program, and order of construction/destruction. `Preallocated Objects` were added to support removal of static objects. Our first `Preallocated Object` was the mutex used for Double-Checked Locking [13] in the ACE implementation of `...` the `Singleton` adapter template.

Current development efforts include breaking the one instance into multiple `Object Lifetime Managers`, to support subsetting. Each layer of ACE, *e.g.*, OS, logging, threads, connection management, sockets, interprocess communication, service configuration, streams, memory management, and utilities, will have its own `Object Lifetime Manager`. When any `Object Lifetime Manager` is instantiated, it will instantiate each dependent `Object Lifetime Manager`, if not already done. And similar, configured-in cooperation will provide graceful termination.

A highly portable implementation of the Object Lifetime Manager pattern and the `Singleton` adapter template is freely available and can be downloaded from `www.cs.wustl.edu/~schmidt/ACE-obtain.html`.

Acknowledgments

Thanks to Matthias Kerkhoff, Per Andersson, Steve Huston, Elias Sreih, and Liang Chen for many helpful discussions on the design and implementation of ACE's `Object Manager`. Thanks to Brad Appleton, our PLoP '99 shepherd and C++ Report Patterns++ section editor, Evgeniy Gabrilovich, Kevlin Henney, and our PLoP '99 workshop group for many helpful suggestions on the content and presentation of the Object Lifetime Manager pattern. And thanks to Bosko Zivaljevic for pointing out that static object construction can extend application startup time.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[3] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[4] J. Vlissides, *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.

[5] D. C. Schmidt, "Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components," *C++ Report*, vol. 11, Sept. 1999.

[6] Wind River Systems, "VxWorks 5.3." http://www.wrs.com/products/-html/vxworks.html.

[7] Integrated Systems, Inc., "pSOSystem." http://www.isi.com/products/-psosystem/.

[8] Bjarne Stroustrup, *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998.

[9] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[10] E. Shea, "Java Singleton," June 7, 2000. `http://www.c2.com/cgi/wiki?JavaSingleton`.

[11] D. McNicol, "Another Java Singleton Problem," June 7, 2000. http://www.c2.com/cgi/wiki?AnotherJavaSingletonProblem.

[12] D. Saks, "Ensuring Static Initialization in C++," *Embedded Systems Programming*, vol. 12, pp. 109–111, Mar. 1999.

[13] D. C. Schmidt and T. Harrison, "Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[14] D. C. Schmidt, "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," *C++ Report*, vol. 11, February 1999.

[15] E. Gabrilovich, "Destruction-Managed Singleton: A Compound Pattern for Reliable Deallocation of Singletons," *C++ Report*, vol. 12, Jan. 2000.

[16] B. Milewski, "Strong Pointers and Resource Management in C++," *C++ Report*, vol. 10, pp. 23–27, Sept. 1998.

[17] B. Milewski, "Strong Pointers and Resource Management in C++, Part 2," *C++ Report*, vol. 11, pp. 36–39,50, Feb. 1999.

[18] P. Sommerland and F. Buschmann, "The Manager Design Pattern," in *Proceedings of the 3rd Pattern Languages of Programming Conference*, September 1996.

[19] T. Cargill, "Localized Ownership: Managing Dynamic Objects in C++," in *Pattern Languages of Program Design 2* (J. M. Vlissides, J. O. Coplien, and N. L. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[20] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

A Detailed Implementation

This section provides an example of a concrete Object Lifetime Manager implementation in C++ described in Section 2.9. It is based on the ACE [9] Object Lifetime Manager implementation. The ACE implementation reveals some interesting design issues. Its most visible purpose is to manage cleanup of singletons at program termination, and create/destroy Preallocated Objects. In addition, it performs other cleanup actions, such as shutdown of services provided by the ACE library, at program termination.

The Object Lifetime Manager Base abstract base class, shown in Figure 1, provides the initialization and finalization mechanisms for an Object Lifetime Manager. Subclasses must specialize and provide implementations, described below.

In addition, Object Lifetime Manager Base supports chaining of Object Lifetime Managers. Object Lifetime Managers are Singletons, each with its own locus of interest. An application may have a need for more than one Object Lifetime Manager, *e.g.*, one per major component. Chaining permits ordered shutdown of the separate components.

Figure 2 shows an example Object Lifetime Manager class. It is a Singleton, so it provides a static instance accessor. In addition, it provides static starting_up and shutting_down state accessors. An enumeration lists identifiers for the Preallocated Objects that it owns.

An interesting detail is the (boolean) reference count logic provided by the derived class init and fini methods. There are several alternatives for constructing an Object Lifetime Manager, discussed in Section 2.9. The reference count ensures that an Object Lifetime Manager is only constructed once, and destroyed once.

The implementations of the instance, init, and fini methods are shown in Figure 3 and Figure 4. The instance method is typical of Singleton instance accessors, but includes logic to support static placement instead of dynamic allocation. In addition, it is not thread safe, requiring construction before the program spawns any threads. This avoids the need for a lock to guard the allocation.

The init and fini methods show creation and destruction of Preallocated Objects, respectively. They show application-specific startup and shutdown code. Finally, they show maintenance of the Object Lifetime Manager state.[8]

[8]This should be moved up to the base class.

```
class Object_Lifetime_Manager_Base
{
public:
  virtual int init (void) = 0;
  // Explicitly initialize.  Returns 0 on success,
  // -1 on failure due to dynamic allocation
  // failure (in which case errno is set to
  // ENOMEM), or 1 if it had already been called.

  virtual int fini (void) = 0;
  // Explicitly destroy.  Returns 0 on success,
  // -1 on failure because the number of <fini>
  // calls hasn't reached the number of <init>
  // calls, or 1 if it had already been called.

  enum Object_Lifetime_Manager_State {
    OBJ_MAN_UNINITIALIZED,
    OBJ_MAN_INITIALIZING,
    OBJ_MAN_INITIALIZED,
    OBJ_MAN_SHUTTING_DOWN,
    OBJ_MAN_SHUT_DOWN
  };

protected:
  Object_Lifetime_Manager_Base (void) :
    object_manager_state_ (OBJ_MAN_UNINITIALIZED),
    dynamically_allocated_ (0),
    next_ (0) {}

  virtual ~Object_Lifetime_Manager_Base (void) {
    // Clear the flag so that fini
    // doesn't delete again.
    dynamically_allocated_ = 0;
  }

  int starting_up_i (void) {
    return object_manager_state_ <
      OBJ_MAN_INITIALIZED;
  }
  // Returns 1 before Object_Lifetime_Manager_Base
  // has been constructed.  This flag can be used
  // to determine if the program is constructing
  // static objects.  If no static object spawns
  // any threads, the program will be
  // single-threaded when this flag returns 1.

  int shutting_down_i (void) {
    return object_manager_state_ >
      OBJ_MAN_INITIALIZED;
  }
  // Returns 1 after Object_Lifetime_Manager_Base
  // has been destroyed.

  Object_Lifetime_Manager_State object_manager_state_;
  // State of the Object_Lifetime_Manager;

  u_int dynamically_allocated_;
  // Flag indicating whether the
  // Object_Lifetime_Manager instance was
  // dynamically allocated by the library.
  // (If it was dynamically allocated by the
  // application, then the application is
  // responsible for deleting it.)

  Object_Lifetime_Manager_Base *next_;
  // Link to next Object_Lifetime_Manager,
  // for chaining.
};
```

Figure 1: Object Lifetime Manager Base Class

```
class Object_Lifetime_Manager :
  public Object_Lifetime_Manager_Base
{
public:
  virtual int init (void);

  virtual int fini (void);

  static int starting_up (void) {
    return instance_  ?
      instance_->starting_up_i ()  :  1;
  }

  static int shutting_down (void) {
    return instance_  ?
      instance_->shutting_down_i ()  :  1;
  }

  enum Preallocated_Object
    {
# if defined (MT_SAFE) && (MT_SAFE != 0)
      OS_MONITOR_LOCK,
      TSS_CLEANUP_LOCK,
# else
      // Without MT_SAFE, There are no
      // preallocated objects.  Make
      // sure that the preallocated_array
      // size is at least one by declaring
      // this dummy.
      EMPTY_PREALLOCATED_OBJECT,
# endif /* MT_SAFE */
      // This enum value must be last!
      PREALLOCATED_OBJECTS
    };
  // Unique identifiers for Preallocated Objects.

  static Object_Lifetime_Manager *instance (void);
  // Accessor to singleton instance.

public:
  // Application code should not use these
  // explicitly, so they're hidden here.  They're
  // public so that the Object_Lifetime_Manager
  // can be onstructed/destructed in main, on
  // the stack.
  Object_Lifetime_Manager (void) {
    // Make sure that no further instances are
    // created via instance.
    if (instance_ == 0)
      instance_ = this;
    init ();
   }

  ~Object_Lifetime_Manager (void) {
    // Don't delete this again in fini.
    dynamically_allocated_ = 0;
    fini ();
  }

private:
  static Object_Lifetime_Manager *instance_;
  // Singleton instance pointer.

  static void *
    preallocated_object[PREALLOCATED_OBJECTS];
  // Array of Preallocated Objects.
};
```

Figure 2: Object Lifetime Manager Class

```
Object_Lifetime_Manager *
Object_Lifetime_Manager::instance_ = 0;
// Singleton instance pointer.

Object_Lifetime_Manager *
Object_Lifetime_Manager::instance (void)
{
  // This function should be called during
  // construction of static instances, or
  // before any other threads have been created
  // in the process.  So, it's not thread safe.
  if (instance_ == 0) {
    Object_Lifetime_Manager *instance_pointer =
      new Object_Lifetime_Manager;

    // instance_ gets set as a side effect of the
    // Object_Lifetime_Manager allocation, by
    // the default constructor.  Verify that . . .
    assert (instance_pointer == instance_);

    instance_pointer->dynamically_allocated_ = 1;
  }
  return instance_;
}

int
Object_Lifetime_Manager::init (void)
{
  if (starting_up_i ()) {
    // First, indicate that this
    // Object_Lifetime_Manager instance
    // is being initialized.
    object_manager_state_ = OBJ_MAN_INITIALIZING;

    if (this == instance_) {
# if defined (MT_SAFE) && (MT_SAFE != 0)
      PREALLOCATE_OBJECT (mutex_t,
        OS_MONITOR_LOCK)
      // Mutex initialization omitted.

      PREALLOCATE_OBJECT (recursive_mutex_t,
        TSS_CLEANUP_LOCK)
      // Recursive mutex initialization omitted.
# endif /* MT_SAFE */

      // Open Winsock (no-op on other
      // platforms).
      socket_init (/* WINSOCK_VERSION */);

      // Other startup code omitted.
    }

    // Finally, indicate that the
    // Object_Lifetime_Manager instance
    // has been initialized.
      object_manager_state_ = OBJ_MAN_INITIALIZED;
      return 0;
    } else {
      // Had already initialized.
      return 1;
    }
}
```

Figure 3: Object Lifetime Method Implementations

```
int
Object_Lifetime_Manager::fini (void)
{
  if (shutting_down_i ())
    // Too late.  Or, maybe too early.  Either
    // <fini> has already been called, or
    // <init> was never called.
    return object_manager_state_ ==
      OBJ_MAN_SHUT_DOWN  ?  1  :  -1;

  // Indicate that the Object_Lifetime_Manager
  // instance is being shut down.
  // This object manager should be the last one
  // to be shut down.
  object_manager_state_ = OBJ_MAN_SHUTTING_DOWN;

  // If another Object_Lifetime_Manager has
  // registered for termination, do it.
  if (next_) {
    next_->fini ();
    // Protect against recursive calls.
    next_ = 0;
  }

  // Only clean up Preallocated Objects when
  // the singleton Instance is being destroyed.
  if (this == instance_) {
    // Close down Winsock (no-op on other
    // platforms).
    socket_fini ();

    // Cleanup the dynamically preallocated
    // objects.
# if defined (MT_SAFE) && (MT_SAFE != 0)
    // Mutex destroy not shown . . .
    DELETE_PREALLOCATED_OBJECT (mutex_t,
      MONITOR_LOCK)

    // Recursive mutex destroy not shown . . .
    DELETE_PREALLOCATED_OBJECT (
      recursive_mutex_t,
      TSS_CLEANUP_LOCK)
# endif /* MT_SAFE */
    }

  // Indicate that this Object_Lifetime_Manager
  // instance has been shut down.
  object_manager_state_ = OBJ_MAN_SHUT_DOWN;

  if (dynamically_allocated_)
    delete this;

  if (this == instance_)
    instance_ = 0;

  return 0;
}
```

Figure 4: Object Lifetime Method Implementations, cont'd.