Destruction-Managed Singleton: a compound pattern for reliable deallocation of singletons

Evgeniy Gabrilovich
gabr@acm.org

Singleton [1] is a creational pattern with well-defined semantics ensuring that the instance is always created prior to use. This effectively solves the problem of initialization order when a number of interrelated objects are involved. But the pattern's destruction semantics is inadequate for several singletons with complex dependencies among them. A *Destruction-Managed Singleton* complements this pattern by imposing a sound order of object destruction. Destruction-Managed Singleton is an instance of the Object Lifetime Manager pattern [2], which "governs the entire lifetime of objects, from creating them prior to their first use, to ensuring they are destroyed properly at program termination".

Intent

Ensure the destruction of interdependent singletons in the correct order, and guarantee that there are no attempts to use previously deallocated objects.

Motivation

For example, suppose there is a global Logger object, and methods of other global entities use it for recording various status messages. Suppose further that the destructors of these entities must notify the Logger about system resources they release. Obviously, the Logger should be the last one to be destroyed. If the language rules destroy the Logger first, other objects might unknowingly attempt to use it, leading to unpredictable (and most probably disastrous) consequences.
Figure 1 illustrates this scenario with an application which has a Logger object and a Resource object (representing another global entity). First, the application obtains a handle to the Resource (via function instance()), and then invokes its function process(). The latter uses a Logger to log a status message (using function log()). At program end, the application first destroys the Logger, then the Resource. The destructor of Resource attempts to use the Logger (calling Logger::instance()), and then all bets are off …

This problem can be solved by using a dedicated Destruction Manager to control the order of singleton destruction. Whenever a singleton is created, its constructor notifies the Destruction Manager of when it should be destroyed (relative to other singletons). The constructor creates a *destructor* object which contains a pointer to the singleton and its (user-assigned) *destruction phase*[1]. The *destructor* is automatically registered with the Destruction Manager, which assumes responsibility for the corresponding singleton from then on. At the end of main(), the programmer should invoke a dedicated function of the Destruction Manager (destroy_objects()). The latter sorts all the *destructors* registered with it in the decreasing order of their phases, and then destroys the singletons in this order. The destruction per se is performed by calling a function destroy() of the *destructor*, which in turn invokes the function

[1] The smaller the phase, the later the singleton should be destroyed. We assume there are no circular dependencies between singletons, so it is possible to assign each one an appropriate destruction phase.

destroy_instance() of the singleton (accessible to the *destructor* due to its friend relationship with the latter).

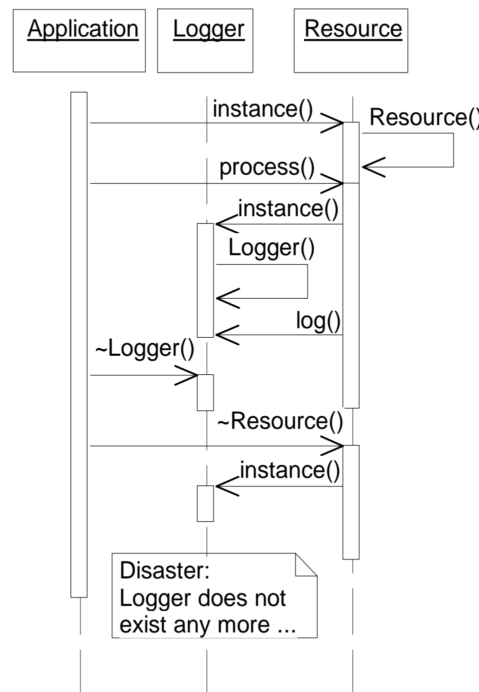

Figure 1. Interaction diagram that illustrates the problem.

From the structural point of view, Destruction-Managed Singleton is a *compound pattern*. It utilizes the reciprocity between Singleton and Destruction Manager, using the notion of *registration*[2]. Composite design patterns (or *compound*[3] *patterns*) [3] are such that their basic building blocks are patterns themselves, rather than objects. The key notion in this definition is the *synergy* between the constituent patterns, which accounts for the ability of individual patterns to work together to become a more useful pattern.

Since all the clients of a Singleton class share its lone instance, the design should impose a controlled protocol regarding the deletion of this object. Moreover, special care must be taken of the destruction order in the presence of several singleton objects depending on each other [6]. The Destruction Manager addresses exactly this situation: objects register with it for subsequent destruction, and each object is only destroyed when it is no longer needed. Observe that the Destruction Manager is not specific about the creation of objects, it only cares about their destruction. In fact, the Destruction Manager and the so-called creational patterns [1] lie on the opposite sides of the object lifetime management spectrum. Incidentally, since a single Destruction Manager is usually sufficient, and it should be globally available for other objects to register, it itself may be implemented as a Singleton.

[2] Though not a pattern in its own right, registration of objects with some distinguished entity is a technique frequently used in patterns (e.g., Observer and registration of Prototypes in Abstract Factory, to name but a few).

[3] The name "compound pattern" is used hereafter, to keep with John Vlissides' current usage of the term.

Applicability

If some global object is a client of another one, the latter may not be destroyed until the former terminates. Otherwise, if the former inadvertently invokes a function of the latter after it ceased to exist, the aftermath may be rather gloomy. The C++ Standard [7] prescribes the order of initialization and destruction only for objects defined in the same translation unit. In all other cases, use Destruction-Managed Singleton for safe deallocation of interdependent global objects.

Structure

Figure 2 represents the class diagram of the Destruction-Managed Singleton.

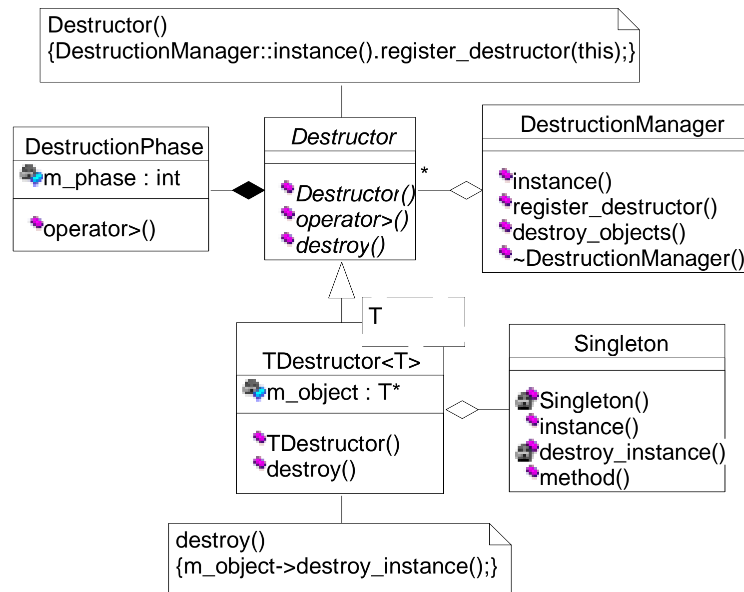

Figure 2. Class diagram for Destruction-Managed Singleton.

Participants

- DestructionManager
  Responsible for destroying registered singletons in user-defined phases.
  - Provides function instance() to access the unique instance (singleton interface).
  - Singleton *destructors* register themselves using the register_destructor() function.
  - The application client invokes function destroy_objects() when graceful shutdown is required.
  - The destructor of the Destruction Manager (~DestructionManager()) deallocates all the *destructor* objects registered with it. As explained above, these auxiliary objects are dynamically created by singleton constructors, and have to be disposed of properly to prevent memory leak.
- DestructionPhase
  Encapsulates the notion of a destruction phase.
  - Destruction phases can be compared using the boolean operator>().
- Destructor
  An abstract base class which represents objects to be destroyed in a particular phase.

- The constructor receives a phase parameter, and registers itself with the Destruction Manager, so that its function destroy() be invoked in this phase.
- Pure virtual function destroy() is overridden in derived classes. Its invocation destroys the object represented by the *destructor*.
- *Destructors* are comparable to one another, based on the values of their phases, via the boolean operator>().

- TDestructor

  A parameterized (template) class, whose instances enclose pointers to actual objects to be destroyed.
  - The constructor receives a pointer to an object and a phase in which the object should be destroyed, and registers with the Destruction Manager due to the implementation of the constructor of the base class.
  - Function destroy() literally destroys the underlying object, by calling its function destroy_instance() (which is assumed to be defined in all the classes instantiating the template).
- Singleton

  In this design, represents a generic singleton object whose destruction should be controlled.
  - instance() is a vanilla access function.
  - The constructor creates a new *destructor* object of type TDestructor<Singleton> to represent this singleton. This *destructor* keeps a pointer to the singleton and the designator of its destruction phase.
  - Function destroy_instance() destroys the singleton.
  - Lastly, method() stands for all the other member functions of the singleton that define its specific behavior. For example, in a Logger class mentioned above, such would be the function void Logger::log(string message), which provides a message logging service for its clients.

Collaborations

The sequence diagram in Figure 3 depicts sample collaborations between the participants of the Destruction-Managed Singleton:

(1) The application requests access to the singleton instance, to invoke its method().

(2) No instance has yet been created, and the static function instance() creates one by calling the singleton constructor.

(3) The Singleton constructor creates a *destructor* object of type TDestructor, to keep the singleton pointer and its destruction phase.

(4) The *destructor* attempts to obtain a reference to the Destruction Manager.

(5) No Destruction Manager exists, so the static function instance() creates one.

(6) The *destructor* registers itself with the Destruction Manager.

(7) The application invokes the method() member function of the singleton.

(8) Toward the end of the program, the application is about to perform a clean shutdown. To this end, it first obtains a reference to the Destruction Manager.

(9) The application calls function destroy_objects() of the Destruction Manager.

(10) The Destruction Manager sorts the *destructors* registered with it in the decreasing order of their phases, using the operator>().

(11) Upon sorting, the Destruction Manager invokes function destroy() of each *destructor*.

(12) The *destructor* forwards the destruction request to the singleton it represents, by calling its function destroy_instance().

(13) The static function destroy_instance() deletes the singleton.
(14) At the end of the program, the destructor of the Destruction Manager is automatically invoked.
(15) The destructor of the Destruction Manager deallocates all the *destructor* objects registered with it.

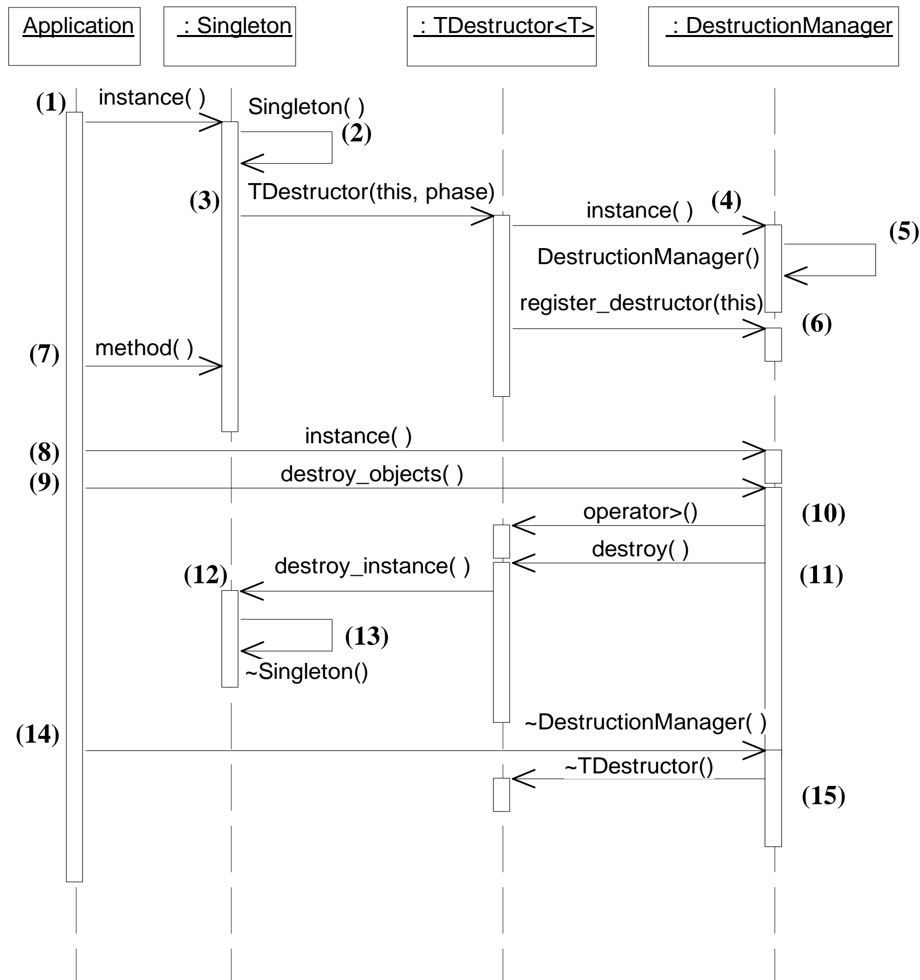

Figure 3. Interaction diagram for Destruction-Managed Singleton.

Implementation

Listings 1 and 2 show the basic Singleton implementation. We use an implementation of Singleton [8] based on the auto_ptr class template – the C++ Standard Library [7] definition of smart pointer. This way, if a singleton is not controlled by the Destruction Manager (e.g., the Destruction Manager is also realized as a singleton, but does not destroy itself), the auto-pointer mechanism destroys the singleton object at program end[4].

The auto_ptr owns the object pointed to by its data member, and is responsible for its deletion. To facilitate this scheme, an auxiliary private function of class Singleton (get_instance()) defines a static auto-pointer to the actual object, which serves as a

[4] In [4], Meyers suggests a Singleton implementation where the object instance is defined static in a dedicated function, which returns a reference to it. Such definition invokes the singleton destructor prior to program termination, thus preventing possible memory and resource leak. Observe that this approach tightly binds the singleton object with the enclosing function.

proxy for the latter. The object instance owned by the auto_ptr is thus detached from the access function wrapper (instance()) that ensures its singleton properties. It is this feature that allows the singleton object to be destroyed either by the auto_ptr itself, or by the Destruction Manager. In the latter case, function auto_ptr<Singleton>::release() is used to retrieve the singleton pointer; it revokes the ownership of the auto_ptr over the singleton, and thus prevents its repeated deletion[5].

The Singleton interface has two public access functions – *const* and *non-const* – which yield the singleton object by dereferencing the auto_ptr. For additional code safety, if the constant object will do the job, the access function const_instance() should be used. Both functions return a reference to the actual object (and not to the auto_ptr), in order to conceal the implementation details from clients.

Listing 1. Singleton implementation (singleton.h):

```
#include <memory>  // for auto_ptr
using namespace std;

class Singleton {
  typedef auto_ptr<Singleton> SingletonPtr;

  // return by reference to prevent transfer of ownership
  static SingletonPtr& get_instance();

  // to allow auto_ptr delete the singleton
  friend class auto_ptr<Singleton>;

  // singletons should not be copied
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);

protected:
  Singleton() { /* initialize the singleton object */}
  ~Singleton() { /* destroy the singleton object */}
public:
  static Singleton& instance() { return *get_instance(); }
  static const Singleton& const_instance() { return instance(); }
};
```

Listing 2. Singleton implementation (singleton.cxx):

```
#include "singleton.h"

Singleton::SingletonPtr& Singleton::get_instance() {
  static SingletonPtr the_singleton(new Singleton);

  return the_singleton;
}
```

Implementation and sample code for the Destruction-Managed Singleton

This section exemplifies the gist of the solution. The complete example code is available at the C++ Report Web site at http://www.creport.com/ …

---

[5] According to the approved C++ Standard [7] (and as opposed to its previous draft editions), the release() function must set the auto_ptr data member pointer to NULL. This prevents repeated deletion when the auto_ptr checks the ownership over the pointed object in its destructor.

Listing 3 defines the destruction phase. Remember that the smaller the phase, the later the object should be destroyed.

Listing 3. Definition of the destruction phase (dphase.h):
```
class DestructionPhase {
  int m_phase;
public:
  explicit DestructionPhase (int phase) : m_phase(phase) {}

  bool operator> (const DestructionPhase& dp) const
  { return m_phase > dp.m_phase; }
};
```

Listing 4 shows the hierarchy of classes for *destructor* objects. Destructor is an abstract class whose instances can be sorted according to their destruction phases. Class template TDestructor<> is derived from Destructor, and contains a pointer to the singleton object it is responsible for. The template parameter is instantiated to actual singleton classes that are assumed to define function destroy_instance(), which allows deletion of the object inside the auto_ptr (that's why it is necessary to detach the object from its wrapper).

Listing 4. Definition of class Destructor and class template TDestructor (destructor.h):
```
#include "dphase.h"
#include "dmanager.h"

class Destructor {
  DestructionPhase m_dphase;
public:
  Destructor(DestructionPhase dphase) : m_dphase(dphase)
  { DestructionManager::instance().register_destructor(this); }

  bool operator>(const Destructor& destructor) const
  { return m_dphase > destructor.m_dphase; }

  virtual void destroy() = 0;
};

template <class T> class TDestructor : public Destructor {
  T* m_object;
public:
  TDestructor(T* object, DestructionPhase dphase)
    : Destructor(dphase), m_object(object) {}

  void destroy() { m_object->destroy_instance(); }
};
```

The Destruction Manager is outlined in Listings 5 and 6 (some obvious details due to the Destruction Manager's being a singleton have been omitted for the sake of brevity). Singletons register their *destructors* with the Destruction Manager via function register_destructor(). The Destruction Manager is responsible for the memory occupied by (dynamically allocated) *destructor* objects, therefore, it has to delete them before it terminates.

Function destroy_objects() sorts the *destructors* in the decreasing order of their phases, and then consecutively destroys the singleton objects they manage. It is this function

that should be manually invoked at the end of main() to ensure proper destruction order of the program singletons. Class template greater_ptr<> is an auxiliary predicate for comparing objects given their pointers.

Listing 5. Destruction Manager (dmanager.h):

```
#include <memory>
#include <vector>
using namespace std;

class Destructor;  // forward declaration
class DestructionManager {
  typedef auto_ptr<DestructionManager> DestructionManagerPtr;

  vector<Destructor*> m_destructors;

  static DestructionManagerPtr& get_instance();

  DestructionManager() {}
  ~DestructionManager();

  friend class auto_ptr<DestructionManager>;

public:
  // singleton interface
  static DestructionManager& instance() { return *get_instance(); }

  void register_destructor(Destructor* destructor)
  { m_destructors.push_back(destructor); }

  void destroy_objects();  // destroy the objects
};
```

Listing 6. Destruction Manager (dmanager.cxx):

```
#include <algorithm>  // for sort()
#include "dmanager.h"
#include "destructor.h"
using namespace std;

DestructionManager::~DestructionManager() {
  for (int i = 0; i < m_destructors.size(); ++i)
    delete m_destructors[i];
}

template <class T> class greater_ptr {
public:
  typedef T* T_ptr;

  bool operator()(const T_ptr& lhs, const T_ptr& rhs) const
  { return *lhs > *rhs; }
};

void DestructionManager::destroy_objects() {
  // sort the destructors in decreasing order
  sort(m_destructors.begin(), m_destructors.end(),
       greater_ptr<Destructor>());

  // destroy the objects
  for (int i = 0; i < m_destructors.size(); ++i)
    m_destructors[i]->destroy();
```

}

It is the responsibility of the programmer to invoke function destroy_objects() on exit from main(). Observe that the destructor of DestructionManager does not call this function, because there is no control of when the destructor is invoked. For example, it may be invoked after the application shutdown process has started, in which case it is too late to call destroy_objects(), since some of the singletons may have already been destroyed.

Listings 7 and 8 show a sample resource definition – a message logging class Logger (again, insignificant details have been omitted). The Logger here corresponds to the generic Singleton class of Sections "Structure", "Participants" and "Collaborations". The Logger constructor creates a new TDestructor object, which contains all the information necessary to destroy the Logger object at the right time. The constructor of TDestructor registers it with the Destruction Manager; the latter ultimately deletes the logger, and then deallocates the *destructor* object itself.
Function destroy_instance() releases the singleton object from the auto_ptr, and then destroys it. Since the auto_ptr no longer owns the object, it would not attempt to delete it when the program terminates. Class TDestructor is defined a friend of Logger, so that its instances may access the function destroy_instance().

Listing 7. Sample resource definition (logger.h):

```
#include <memory>
#include <string>
#include <stdexcept>
#include "dphase.h"
#include "destructor.h"
using namespace std;

class Logger {
  typedef auto_ptr<Logger> LoggerPtr;

  static LoggerPtr& get_instance();
  static void destroy_instance() { delete get_instance().release(); }

  Logger();
  ~Logger();

  friend class auto_ptr<Logger>;
  friend class TDestructor<Logger>;

public:
  // checked singleton interface
  static Logger& instance() throw (std::logic_error) {
    Logger* logger = get_instance().get();
    if ( !logger )
      throw logic_error("Logger is not available!");
    return *logger;
  }

  void log(string message);
};
```

Note the implementation of the instance() function: it does not immediately return the dereferenced auto_ptr (in contrast to Listing 1), but first checks if it still points to a valid object. This test may fail in either of the following two cases:

1) memory allocation has failed[6] in function get_instance() (see Listing 8), or
2) the singleton object has already been destroyed.

The former case is extremely rare. The latter may occur if incorrect destruction phases have been assigned in the program, and some singleton is deleted prior to its last use. In such a case, when the Destruction Manager destroys the singleton via function release() of the auto_ptr, the data member pointer of the latter is cleared (see footnote 5), and this causes the validity test to fail. Although this test incurs a slight run-time penalty on each singleton access, it should be used at least during the debugging stage, to verify the destruction policy. The idea of checking singletons for prior deletion comes from [11].

A word of caution: if a singleton's destructor invokes a member function of another singleton that has already been destroyed, an std::logic_error exception will be thrown. It is dangerous for this exception to leave the destructor, for if the latter was invoked during stack unwinding due to an exception thrown earlier, function terminate() will be immediately called, aborting the application [7]. This situation is discussed in [5].

Listing 8. Sample resource definition (logger.cxx):

```
#include <iostream>
#include <string>
#include "dphase.h"
#include "destructor.h"
#include "logger.h"
using namespace std;

Logger::LoggerPtr& Logger::get_instance() {
  static LoggerPtr the_logger(new Logger);
  return the_logger;
}

Logger::Logger() {
  new TDestructor<Logger>(this, DestructionPhase(1));
  cout << "Logger created" << endl;
}

Logger::~Logger() { cout << "Logger destroyed" << endl; }

void Logger::log(string message)
{ cerr << "Logger: " << message << endl; }
```

The complete example code (available from the C++ Report Web site) also defines class Resource, whose destructor uses function Logger::log() to record error messages (if any).

At last, Listing 9 presents the function main().

Listing 9. Sample main program (main.cxx):

```
#include "dmanager.h"
#include "resource.h"
```

[6] Unless the operator new throws on failure an std::bad_alloc exception.

```
class DestroyObjects {
public:
  ~DestroyObjects()
  { DestructionManager::instance.destroy_objects(); }
};

void main(void) {
  DestroyObjects destroyer;

  // ...
  Resource::instance().process();

  // The destructor of 'destroyer' is invoked automatically on exit
  // from 'main()', commencing the controlled destruction of singletons.
}
```

The function DestructionManager::instance.destroy_objects() is invoked by the destructor of a utility class DestroyObjects, thus relieving the programmer from having to call it explicitly at the end of main(). This approach facilitates multiple exit points from main(), so it is not necessary to copy the cleanup code over and over again. More important, it also works in case of exceptions propagating through main(), since destructors of local objects are automatically invoked during stack unwinding. If destroyer is the first local variable defined in main(), its destructor would be called last [7], and thus the Destruction Manager would be invoked immediately before leaving the function.
In the example code available at the C++ Report Web site, class DestroyObjects is defined in file "dmanager.h", following the definition of class DestructionManager. This is more convenient to clients, as the Destruction Manager comes complete with this auxiliary class.

Consequences

Destruction-Managed Singleton is a compound pattern for controlling the destruction order of singleton objects. It achieves this aim due to the cooperation of the following individual patterns and techniques:

- The *Singleton* pattern resolves the initialization order of interdependent global objects, and ensures the program-wide uniqueness of the resource it manages.
- The *Destruction Manager* is a complementary pattern for managing the other end of objects' life span, namely, destruction. Therefore, it may be considered a *destructional* pattern, as opposed to creational patterns [1] like Abstract Factory, Factory Method, or Singleton itself for that matter.
- The *Proxy* pattern (realized through the auto_ptr implementation of the concept of *smart pointer*) allows to detach the Singleton object from its wrapper, and thus facilitates its destruction by the Destruction Manager (bypassing the regular auto_ptr mechanism).
- The *Registration* technique enables singletons to sign up with the Destruction Manager for subsequent destruction at an appropriate time.

The Destruction-Managed Singleton handles the entire life span of a singleton, extending the behavior of the original pattern [1]. This compound pattern guarantees the object is created prior to use, and exists as long as it is needed. In addition to enforcing the destruction policy, the Destruction-Managed Singleton constantly verifies its validity, and throws an exception whenever a sound destruction order is breached.

Discussion

This section examines several incidental issues, including the tradeoffs incorporated in the Destruction-Managed Singleton.

Managing non-singleton objects

When a program uses global objects which are not singletons, and especially if there are singletons that depend on non-singletons, the more comprehensive Object Lifetime Manager [2] should be used. This pattern controls creation and destruction of objects which are not necessarily singletons, and is available as a built-in part of ACE [9].

Alternative destruction policies

Instead of using an express notion of destruction phases, an alternative approach could require each singleton to explicitly specify on which other objects it depends. The Destruction Manager would then perform a topological sort of the dependencies graph, deducing the phases of destruction automatically. This would require the constructor of each singleton to notify the Destruction Manager of all the other singletons it depends on. In a simple solution where the constructor registers pointers to all the other singletons it might use, all those would be created even if some of them are not needed in a particular program execution. A more elaborate solution would identify singletons by their (unique) string names rather than pointers, but this seems to be an overkill.

Thread-safety

Among the issues this article does not address is thread-safety. Probably every existing singleton implementation uses variables defined at global scope. In a multi-tasking environment this may constitute a problem, should a singleton have to serve multiple threads. The Double-Checked Locking pattern [10] presents a good solution to this problem, minimizing the amount of coordination necessary for preserving the consistency of critical sections. Vlissides [6] notes that in multi-threaded applications it would be well-advised to use an access function `instance()` that returns a pointer to the singleton rather than a reference. This is because “some C++ compilers generate internal data structures that cannot be protected by locks”.

Genericity

Finally, it should be mentioned that ACE [9] provides a reusable singleton adapter, `ACE_Singleton`, which accepts a user-defined class as a parameter, and makes it a Singleton. The idea is based upon the concept of separation of responsibilities between the two classes: the user class (also known as the adapted class) and the adapter class. The latter is solely responsible for the "Singleton-ness" of the former, which can concentrate on the business modeling aspects per se. The Double-Checked Locking technique [10] is already incorporated in this adapter, which contains an additional parameter to facilitate different locking policies. If necessary, the `ACE_Singleton` template can be easily integrated into the Destruction-Managed Singleton.

Acknowledgments

Destruction-Managed Singleton was inspired by an example code from [12], where an initialization manager working in phases was used to resolve the mutual initialization order of global variables (without singletons, but using two-stage object construction). Special thanks are due to Brad Appleton, Patterns++ Section Editor, for many enlightening discussions and for his guidance during the preparation of this article. Thanks to Avner Ben and Vitaly Surazhsky for their constructive comments and suggestions.

References

[1] Gamma, E., *et al.* "Design Patterns: Elements of Reusable Software Architecture", Addison Wesley, 1995.

[2] Levine, D. L., *et al.* “Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction”, C++ Report, 12(1), January 2000.

[3] Vlissides, J. "Composite Design Patterns", C++ Report, 10(6): 45-47, June 1998.

[4] Meyers, S. "Effective C++", 2nd edition, Addison Wesley, 1998.

[5] Meyers, S. "More Effective C++", Addison Wesley, 1996.

[6] Vlissides, J. "Pattern Hatching: Design Patterns Applied", Addison Wesley, 1998.

[7] "Information Technology – Programming Languages – C++", International Standard ISO/IEC 14882-1998(E).

[8] Gabrilovich, E. “Controlling the Destruction Order of Singleton Objects”, C/C++ Users Journal, October 1999.

[9] Schmidt, D. C. “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] Schmidt, D. C. and T. Harrison “Double-Checked Locking – An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects”, in *Pattern Languages of Program Design* (Martin, Buschmann, and Riehle, eds.), Addison Wesley 1997.

[11] Alexandrescu, A. Private correspondence, 1999.

[12] Ben-Yehuda, S. "C++ Design Patterns" course, SELA Labs (http://www.sela.co.il).

About the author

Evgeniy Gabrilovich is a Strategic Development Team Leader at Comverse Technology Inc., a developer of multimedia communications processing technology. He holds an M.Sc. in Computer Science from the Technion - Israel Institute of Technology. His interests involve Natural Language Processing, Artificial Intelligence, and Speech Processing. He can be contacted at gabr@acm.org .