JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002
Vol. 1, No. 2, July-August 2002

Thread specific singletons: Handling singleton pattern errors in multi-threaded applications and their variations

Yagna Pant Yagna.Pant@motorola.com
Kazuhiro Ondo Kazuhiro.Ondo@motorola.com
Motorola, Arlington Heights, IL, USA.

The use of the design patterns has been popularized with the publication of the Design Patterns book by Gamma et al. (1995). In this paper we discuss the implementation of the singleton design patterns from "error recovery" perspectives for multi-threaded applications where dynamic memory is allocated on a task by task basis. The description of the singleton pattern is straight forward but its implementation issues are complicated [Alexandrescu, 2001]. We highlight a few problems of traditional implementation for multi-threaded applications and propose several alternatives that could potentially be used in different contexts depending on the problem at hand. Because of the popularity of the C++programming languages in the industry, we also present sample implementation details in C++.

1 THE SINGLETON PATTERN

In real world applications there are many situations where there can be *one and only one* instance of a class. Typical examples may include a class for logging events to a serial output device, a class for authenticating and verifying passwords entered in a website, etc. The purpose of the singleton pattern, in the context of object-oriented programming, is to provide a generic mechanism such that the class is instantiated only once. Normally the first invocation of the class *creates* the unique instance of the class and subsequent invocations would simply return a reference[1] to the instance created earlier. Figure 1 represents the Object Model Diagram (OMD) (after Gamma et al., 1995) of singletons implemented in the *traditional*[2] way (e.g. Figure 2).

As shown in the code fragment (Figure 2), the mechanism for implementing the

[1]The term reference in this context is used generically unless stated otherwise. But this may be implemented as a pointer in C++.

[2]We distinguish between traditional implementation with the extensions presented in this paper.

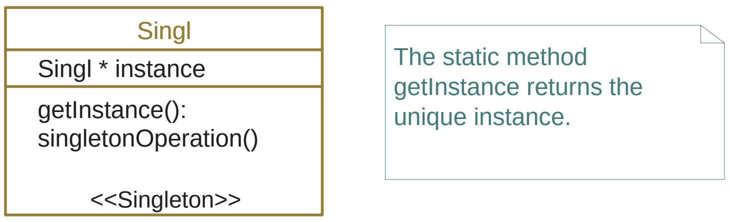


Figure 1: Object model diagram of the singleton pattern

```
class Singl {
public:
  // method for accessing the one and only one instance
   static Singl *getInstance( );

   // a set of operations on the singleton (not defined)
private:
   // Don't allow ourselves to be instantiated and
   // copy constructed outside this class
   Singl( ){ }
   Singl( const Singl & ) { }
   // variable to keep track of the instance
   static Singl *instance;
};
```

```
Singl* Singl::instance = 0;

Singl* Singl::getInstance( ){
   if( 0 == instance ) {
      instance = new Singl;
   }
   return instance;
}

int main( )
{
   Singl *singlPtr = Singl::getInstance( );

   // operations on singlPtr
   return 0;
}
```

Figure 2: Typical implementation of the singleton pattern in C++

singleton pattern is quite straight forward in C++. A class specific `static` variable `instance` is defined to control creation of more than one instances and the `static` member function `getInstance` is used for accessing the one and only one instance of the singleton class[3]

Initially the `static` class variable `Singl::instance` is initialized to `0` (by default) and when the `static` function `Singl::getInstance` is invoked (in the `main` routine), an instance of the `Singl` class is `new`ed in the heap and the `instance` pointer is returned to the application. Normally constructors and copy constructors of the singleton class are made `private` or `protected` such that an attempt to instantiate the `Singl` (class on its own) is flagged as an error during compilation.

2 SINGLETON PROBLEMS

The code given in Figure 2 works well for a single threaded application. If there are a number of tasks or threads accessing the singleton object/class (O/C) in an application, the first task executing the singleton O/C allocates memory for the

[3]The main objective of the paper is to analyze some of the error scenarios in multi-threaded applications. Thus to illustrate the concepts only, the implementation has been kept very simple and intentionally ignores some of the consequences of the Singleton Pattern [Gamma et al., 1995] such as refinements of operations and representation.

instance variable. Since subsequent singleton access within this task and other tasks[4] are *read only*, the above mechanism also works for a *well-behaved* multi-tasking application.[5] However the code given in Figure 2 can create problems if the singleton pattern is in a layer which is common to a number of tasks as shown in Figure 3.

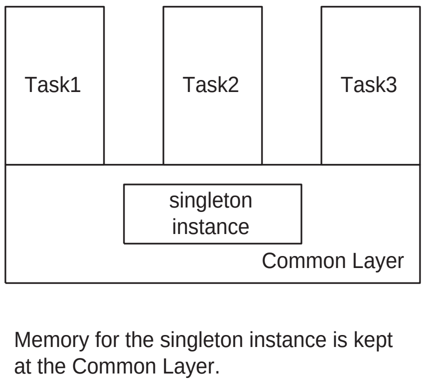

Task1
Task2
Task3
singleton instance
Common Layer
Memory for the singleton instance is kept at the Common Layer.

Figure 3: A multi-tasking software layer where singletons are implemented in the common layer

In Figure 3, there are three tasks which use the services of the underlying lower (or common) layer. Let us assume that the common layer has a single O/C implementing the singleton pattern as shown in Figure 2. The layering approach illustrated in Figure 3 is quite common in many applications (including real-time/embedded systems) whereby the upper layers use the services provided by the lower layers. In an object-oriented application, the lower layers could be a generic framework or a set of reusable class libraries. Let us further assume that the tasks Task1, Task2, and Task3 all use the singleton instance implemented in the common layer. Let us now examine a few situations while the application is being executed:

- Task1 starts executing first and attempts to get an instance of Singl from the common layer. Since the instance pointer initially points to 0 memory (Figure 2) and since no instance has been created, an instance of the Singl is allocated in Task1's heap and the pointer is stored in the class static attribute instance.
- After the context switch, Task1 is suspended and Task2 becomes active. It now needs an instance of the Singl class and invokes the method Singl::getInstance. Since memory has already been allocated and it is not pointing to 0, it simply returns the address stored in the Singl pointer instance.

[4]Here we are assuming that the memory allocated in the heap by one task is accessible to other tasks as well and the dynamic memory is allocated per task basis. In other words each task has its exclusive pool of dynamic memory. Furthermore it is assumed that the singletons are defined in a common lower layer (such as a library).

[5]We have not examined the effect of multiple processors and parallel processing in this paper.

In this paper our main focus is on a unique kind of application architecture. In this architecture, an application can have many tasks or threads (light weight processes). Tasks are identified by a unique set of numbers. When the application is started, the application simply creates the main task or the "master task". Rest of the tasks are created by sending messages to the main task and specifying the entry points, heap size, etc. As all tasks are started one at a time, the double check guard mechanism [Schmidt, 1999] to handle the issue of thread safe initialization (multiple threads executing on a parallel machine when creating the unique instance) was not necessary for our application domain.

All of the tasks running on the board need to be registered and pre-configured with an internal "master task". The "master task" takes care of many details such as starting the new task, allocating dynamic memory space for each tasks and ending the task (either by sending a message explicitly or if the "master task" detects an internal error via. some pre defined mechanisms. There is no shared memory between tasks and communications among tasks is strictly via. message routing using a message queue. Resource congestion/deadlocks (memory management, etc.) are handled at a lower layer using semaphores and application do not need to worry about them. A task can not start another tasks (this is the responsibility of the "master task" and a task can have a number of instances running on the same board. The application architecture has been designed such that when a task is terminated, it won't have any effects to the rest of the tasks running on the board. More details are discussed in [Pant, 2001].

Figure 4: Description of the application architecture

- Similarly when `Task3` starts to execute and needs to refer to the `Singl` class, the same pointer `instance` is returned to the task.

So far so good. All the tasks are behaving properly and all tasks can access the global `static` variable pointer `instance`. Now because of some problems, let us assume that `Task1` starts to misbehave and is terminated.[6]

When a task is terminated, the heap memory associated with that task would no longer be valid and is likely to contain an invalid address.[7] This would cause the state of the singleton class to be altered. Since the memory for the singleton `instance` comes from `Task1`'s address space, an attempt to access the singleton class either by `Task2` or `Task3` is likely to cause a core dump or a bus error. This may bring down the whole application.

[6]Errors in a task means memory problems (within that task), division by zero, or other internal failures. Normally when a task within an application encounters a problem, the entire application will be in an undefined state. However our application does not use any shared memory and there is no other coupling between any two tasks. All tasks have their own heap, their own queues for receiving messages and they communicate with each other by sending messages. Thus our aim is to create tasks that are potentially isolated from each other.

[7]As discussed earlier, in our software architecture, each task has its associated memory heap which is preallocated when the task is started. When the task is killed and restarted, the singleton pointers for that task would be initialized at the beginning (discussed below).

There may be a number of situations where it may not cause any catastrophic effects when the problems in one task propagate to the rest of the application. However this behavior would not be normally tolerated in a mission-critical, multi-tasking applications such as a wireless switch. In such systems it is desirable to confine all problems to the task it self as far as possible such that the errant task can be restarted again to resume the operation.

3 POSSIBLE ENHANCEMENTS

Figure 5 shows one possible strategy to fix the problem [Alexandrescu, 2001] of the singleton pattern as discussed in the last section. The `static` pointer variable `instance` of Figure 2 has been removed and is replaced by an instance `singl` of the class `Singl`. Thus an instance of the `Singl` class would automatically be created when the code object is executed/downloaded in the default data segment. The `getInstance` method returns the address of the `singl` object. Since `static` variables are initialized only once, subsequent invocations of the `getInstance` method would not create another instance of the `Singl` class.

```
class Singl {
public:
   // method for accessing the one and only one instance
   static Singl *getInstance( );

   // a set of operations on the singleton (not defined)
private:
   // Don't allow ourselves to be instantiated and
   // copy constructed outside this class

   Singl( ){ }
   Singl( const Singl & ) { }
};
```

```
Singl* Singl::getInstance( ){
   static Singl singl;

   return &singl;
}

int main( )
{
   Singl *singlPtr = Singl::getInstance( );
   Singl *singlPtr2 = Singl::getInstance( );

   // operations on singlPtr
   return 0;
}
```

Figure 5: An alternative way of fixing the singleton problem using global memory

The above approach may work in an operating system environment which has flat memory address space and global data segment can be accessed by all of the tasks/threads. But this may create some problems in an operating system architecture where by global data is not allowed and individual tasks have corresponding heap and data segment exclusively for themselves. Furthermore, this approach may also have performance related issues as illustrated in Figure 6. There are three tasks and three possible message queues in Figure 6. If there is a single class, all of the messages posted to *all* of the queues is routed via. this class and a filtering mechanism needs to be implemented in the singleton class to route the messages to different tasks appropriately. If the singleton class also implements a *publish–subscribe* design pattern and if the message filtering is done per message type, this list could potentially be huge and there may be issues with traversing the list.

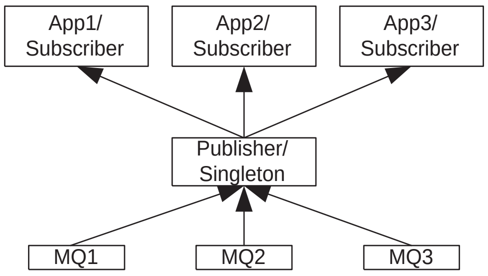
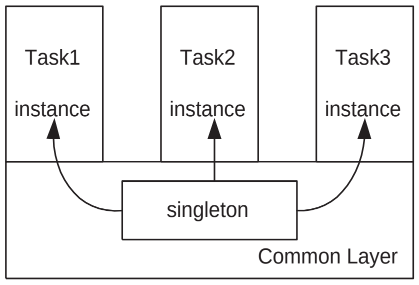


Figure 6: Performance problem of the global instance



Memory for the singleton instance is kept at the task specific heap, whereas the code resides in the Common Layer.

Figure 7: A software architecture for fixing the singleton problems

Another alternative fix would be to create a task specific array of memory (at the stack) and associate this memory with singleton instances as shown in Figure 7 where the code that implements the singleton pattern resides in the common layer, whereas the memory for the singleton `instance` is allocated in each task's address space. There are three pointers (one per each task) pointing to the singleton class and share the same logic for the access operation from the common layer. If the task e.g. `Task1` dies, then only the instance pointer associated with this task would point to an invalid address, without affecting the other two tasks and the rest of the application. Thus in summary the problems caused by any one task would be localized and would not be propagated to the rest of the tasks/applications.

One of the possible implementation strategy for the singleton architecture of Figure 7 is shown in Figure 8. As shown in Figure 8, the `enum SinglEnum` lists all of the singletons across the application.[8] Both the declaration and the definition of the `static` variable `instance` have been removed from the `Singl` class. The main entry point of each task defines an array big enough to hold pointers to the instances of the singletons (for that task). Thus the singletons now use stack memory from each tasks address space. This memory address is then associated with the current

[8]It should be worthwhile to mention that type enumerations could be a potential source of maintenance problems as new singletons are created within the common framework.

```
extern void *staticBufferPtr;

typedef enum{
   MY˙SINGLETON,
   //... indices to other instances of singletons
   MAX_NUM_OF_SINGLETONS
} SinglEnum;

class Singl {
public:
   // method for accessing the one and
   // only one instance
   static Singl *getInstance( );
   // a set of operations on the singleton
   // (not defined)
private:
   // Don't allow ourselves to be instantiated
   // outside this class
   Singl( ){ }
   Singl( const Singl & ) { }
};

Singl* Singl::getInstance( ){
   if( 0 == (Singl*)staticBufferPtr[MY_SINGLETON]){
     staticBufferPtr[MY_SINGLETON]=(void*) new Singl;
   }
   return (Singl*) staticBufferPtr[MY_SINGLETON];
}
```

```
void *staticBufferPtr;

void Task1EntryPoint( )
{
   unsigned void *
     staticBuffers[MAX_NUM_OF_SINGLETONS];
   staticBufferPtr = staticBuffers;

   // add the task specific varaiable list here

   Singl *singlPtr = Singl::getInstance( );

   // operations on singlPtr
}

void Task2EntryPoint( )
{
   unsigned void *
  staticBuffers[MAX_NUM_OF_SINGLETONS];
   staticBufferPtr = staticBuffers;

   // add the task specific varaiable list here

   Singl *singlPtr = Singl::getInstance( );

   // operations on singlPtr
}
```

Figure 8: The memory for the static instance variable is allocated from the task's address space

task such that this memory is used (after context switch) when the singleton code is executed.[9] In summary, by using this approach, we are creating `instance` pointers equal to the number of unique tasks accessing the singletons and associating them with the tasks appropriately.

Another alternative implementation strategy would be to create a *Manager* class that would be responsible for managing the singletons throughout the application. The object model diagram for such an arrangement is shown in Figure 9. The class `ISingl` is the interface class and all singletons throughout the application inherit from this class. The class `SinglMgr` manages instances of the `ISingl` class pointers. `SinglMgr` is a `friend` to `Singl` class such that it can access appropriate constructors and destructors.

The code fragment shown in Figure 10 implements the OMD of Figure 9. As in Figure 8, the code fragment defines an `enum` of singletons used in the application. It then defines an interface class (`ISingl`) with a virtual destructor. The advantage of adding the interface class `ISingl` is in the destruction of the singletons (discussed later). The singleton Manager `SinglMgr` maintains a list of the singletons (pointer to `ISingl` class). The memory for this list is defined at the `main` entry point on the stack. The `SinglMgr` class has a `getInstance` method that takes the singleton index

[9]This feature is available in vxWorks and there should be similar mechanisms in other operating systems as well.

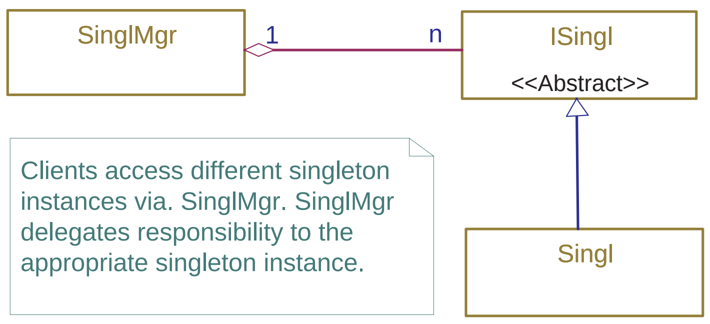

SinglMgr
1
n
ISingl
<<Abstract>>
Singl
Clients access different singleton instances via. SinglMgr. SinglMgr delegates responsibility to the appropriate singleton instance.

Figure 9: OMD for an alternative extension

```
#include <iostream.h>
typedef enum {
  MY_SINGLETON,
  //... indices to other instances
  // of singletons
  MAX_NUM_OF_SINGLETONS
} SinglEnum;

class SinglMgr;

// define an interface class
class ISingl {
  friend SinglMgr;
protected:
  virtual ~ISingl() { }
};

class SinglMgr{
public:
  SinglMgr(ISingl **addr, int sz);
  ~SinglMgr();
  ISingl *getInstance(SinglEnum index) const
    { return sglPtr[index]; }
private:
  ISingl **sglPtr;
  int size;
};


class Singl: public ISingl {
  friend SinglMgr;
public:
  // a set of operations on the singleton
  // (just an example)
  void printHelloWorld() {
    cout << "Hello world" << endl;
  }
private:
  ~Singl() { }
  // Don't allow ourselves to be
  // instantiated outside this class
  // except friends
  Singl( ){ }
  Singl( const Singl & ) { }
};

SinglMgr::SinglMgr(ISingl **addr, int sz) {
  size = sz;
  sglPtr = addr;

  sglPtr[MY_SINGLETON] = new Singl;
  // ....
}

SinglMgr::~SinglMgr() {
  for ( int i = 0; i ¡ size; i++ ) {
    delete sglPtr[i];
  }
}

int main( ) {
  ISingl *staticBuffers[MAX_NUM_OF_SINGLETONS];
  SinglMgr mySinglMgr
     (staticBuffers, MAX_NUM_OF_SINGLETONS);

 // add this to the task var list (not shown)

  Singl * singlPtr =
   (Singl *)mySinglMgr.getInstance(MY_SINGLETON);

  singlPtr->printHelloWorld();
  return 0;
}
```

Figure 10: Singletons are managed by a Manager class

from the `enum` and returns a `ISingl` pointer which can be `cast`ed to an appropriate type (derived class).

As mentioned earlier all of the singletons (e.g. `Singl`) publicly inherit from the `ISingl` class and define `SinglMgr` as a `friend` to `ISingl`. The destructor is `virtual` (to ensure correct destruction) but protected (defined in the `ISingl` class) such that the memory is freed appropriately when `ISingl` pointers are used (within the `SinglMgr` object) to delete child singleton objects[10] and to allow other classes to derive from `ISingl`. As in the earlier examples, the constructors are in the private section. Furthermore, this class does not have any methods to get the singleton instance. The responsibility of getting an instance of all the singletons has been delegated to the `SinglMgr` class.

The constructor of the `SinglMgr` class does a `new` to instantiate individual singletons. Thus the construction order (as well as destruction) of singletons can be explicitly specified if necessary. The destructor iterates through the array to delete singleton instances. Alternatively the destructor can be modified to specify the dependency order in which singletons need to be destroyed (e.g. one singleton depends on the other singleton) without impacting the rest of the application.

The `main` routine defines an array of `ISingl` pointers (in its stack) sufficient to hold a pointer to all singletons. It then instantiates `SinglMgr` class[11] and passes the address of the array as the constructor parameter. The next line shows how to get an instance instance of a singleton.

The constructor of the `SinglMgr` class initializes the singleton pointers by creating a new `instance` explicitly. Thus as long as the `SinglMgr` constructor is invoked successfully, it can be guaranteed the existence of the singleton instances. Hence the `SinglMgr::getInstance` method does not check whether the singleton `instance` is pointing to a `NULL (0)` object. As a result this implementation approach may be slightly faster than the earlier proposal, specially if singletons are accessed quite frequently in an application. As an added advantage, the singletons would be automatically deleted when the `SinglMgr` object goes out of scope.

The code fragment presented in Figure 10 may be suitable for some application areas where the number of singletons are small and known before hand and both the `SinglMgr` and `ISingl` are in the same package. If they are in different packages (e.g. a `ISingl` and `SinglMgr` in a library and `Singl` in application code), this implementation strategy introduces circular dependencies as `SinglMgr` needs to have a knowledge of `Singl`. As the `SinglMgr` is responsible for managing all of the singletons in an application, adding a new singleton requires modification of this class. There is also a loss of type information of "application singletons" in the `getInstance` method as the `SinglMgr` stores an array of `ISingl` object pointers.

One method of overcoming several problems of Figure 10 is illustrated in Figure 11 (OMD) and the corresponding C++code is in Figure 12. In this method the library or the common layer defines the classes `base_singleton, id, singleton`

[10]The destructor should not be made `public` as it allows clients to destroy singleton objects. The destructors of `Singl` should be `protected` or `private` (if it is necessary to prevent other classes to derive from `Singl`).

[11]`SinglMgr` is not a singleton and different tasks can create unique instances of the manager in their main entry routines. As a result when one task is killed, it is not going to effect the rest of the application/tasks.

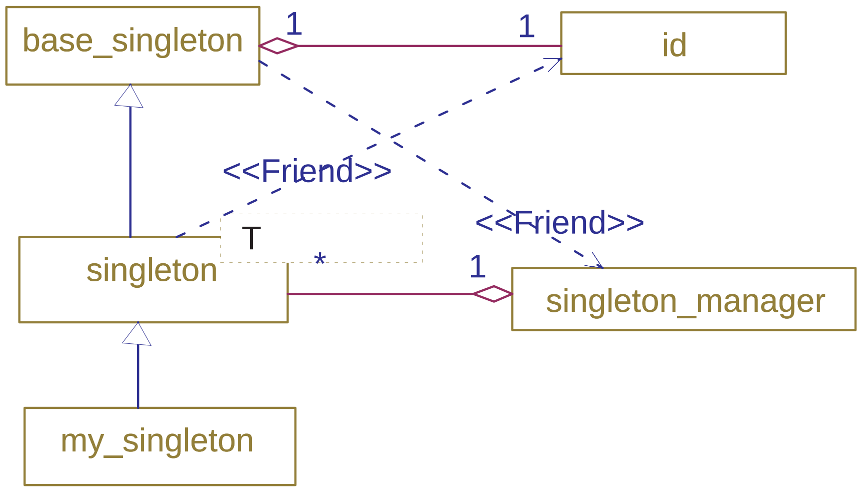


Figure 11: Template based singleton manager

and `singleton_manager`. The class `singleton` is a template class (`T` being the template parameter and inherits from `base_singleton`. There is a static member `id_` inside the `singleton` class for *each* type of application specific singletons (e.g. `my_singleton`). In other words, given two application specific singleton classes `singl1` and `singl2`, both `singleton<singl1>::id_` and `singleton<singl2>::id_` exist and are two distinct objects with distinct values. The class `singleton_manager` holds an array of `singleton`s. It has a member function `template <class T> static T& instance( T* = 0 );` The compiler generates the requested overloaded `instance` functions when needed[12] The class `base_singleton` contains an `id` class (one for each application specific singletons (e.g. `my_singleton`) to determine whether the singleton has already been instantiated.

[12]Since functions can't be overloaded by return type in C++, the function takes a dummy argument which is not used within the member function. Given the two application specific singleton classes `singl1` and `singl2`, the compiler generate the functions `singl1& singleton_manager::instance(singl1* = 0)` and `singl2& singleton_manager::instance(singl2* = 0)` when needed.

```
#include <exception>
#include <vector>

class singleton_manager;
class singleton_error : public std::exception {
public:
   virtual const char * what() const throw() {
      return "singleton error";
   }
};

class base_singleton {
protected:
   class id {
   public:
      id(void) : index(-1) {}
   private:
      int index;
      friend class singleton_manager;
   };
   base_singleton(void) { }
   virtual ~base_singleton(){ }
   void reg_singl(id& id_arg);
   void unreg_singl(id& id_arg);
   friend class singleton_manager;
private:
   // copy-constructor and copy-assignment made
   // private and not implemented, to prevent
   // accidental usage
   base_singleton(const base_singleton&);
   base_singleton& operator=(const base_singleton&);
};

template <class T>
class singleton : public base_singleton {
protected:
   singleton(void) {
      reg_singl(id_);
   }
   virtual ~singleton() {
      unreg_singl(id_);
   }
private:
   friend class singleton_manager;
   static id id_;
private:
   singleton(const singleton&);
   singleton& operator=(const singleton&);
};

class singleton_manager{
public:
   // This class doesnot have any public
   // constructors as there is no need for
   // for singleton_manager objects
   template <class T>
   static T& instance(T* = 0);

private:
   friend class base_singleton;

   static void reg_singl(base_singleton& singl_arg,
   base_singleton::id& id_arg);
   static void unreg_singl(base_singleton& singl_arg,
      base_singleton::id& id_arg) {
      singletons[id_arg.index] = 0;
   }

private:
   static std::vector<base_singleton*> singletons;
private:
   singleton_manager(const singleton_manager&);
   singleton_manager& operator=
                  (const singleton_manager&);
};

inline void base_singleton::reg_singl(id& id_arg){
   singleton_manager::reg_singl(*this, id_arg);
}

inline void base_singleton::unreg_singl(id& id_arg){
   singleton_manager::unreg_singl(*this, id_arg);
}

template <class T>
base_singleton::id  singleton<T>::id_;

void singleton_manager::reg_singl(
  base_singleton& singl_arg, base_singleton::id& id_arg){
    // this function should probably be made multi-
thread safe !!!
   if(-1 == id_arg.index) {
      singletons.push_back(&singl_arg);
      id_arg.index = singletons.size() - 1;
      return;
   }
   if(singletons[id_arg.index])  {
      // this singleton type is already registered
      throw singleton_error();
   }
   singletons[id_arg.index] = &singl_arg;
}
```

(to be continued on next page)

With this approach as the list of singleton objects (and their corresponding pointers/aliases) are automatically maintained in a vector list, the code is easier to maintain. This also removes circular dependency as the singletons are created and added to the list automatically/dynamically within the application domain. Furthermore, this solution removes the possibility of having a mismatch between `static_cast` and the list of enumerations (by application developers). The implementation uses Curiously Recurring Template Patterns [Coplien, 1995], which gives the base class a way of knowing a derived class at compile time (type information is not lost) without creating circular compile time dependencies. However on the

(continued from previous page)

```
template <class T>

T& singleton_manager::instance(T*){
   if(-1 == singleton<T>::id_index)   {
      // no singleton instance registered for this class
      throw singleton_error();
   }
   if(singletons[singleton<T>::id_index]) {
      return dynamic_cast<T&>
      (*singletons[singleton<T>::id_index]);
   }
   // a singleton instance has been registered,
   // but has been deleted
   throw singleton_error();
}

std::vector<base_singleton*>
singleton_manager::singletons;


// application level code

// my_singleton1 is the interface for the
// application
class my_singleton1 : public
singleton<my_singleton1>
{
public:
   my_singleton1(void){ }

   virtual void hello_world(void) = 0;
};
```

```
// my_singleton1_impl is the actual
implementation
class my_singleton1_impl : public
my_singleton1
{
public:
   my_singleton1_impl(void) { }

   virtual void hello_world(void);
};

void my_singleton1_impl::hello_world(void)
{
}

int main(int /*argc*/, char* /*argv*/[])
{
   // scoped singleton
   my_singleton1_impl  ms1;

   // Please note:
   // No identifier, which has to match the
   // specific singleton type and
   // to be maintained manually.
   //
   // No loss of type information
   //
   // The compiler handles it, at a very little
   // runtime overhead during single
   singleton_manager::

instance<my_singleton1>().hello_world();

   return 0;
}
```

Figure 12: Code fragment for implementing the OMD of Figure 11

down side, this implementation strategy has memory overhead from the class `id`. The use of `dynamic_cast` trades some run-time performance overhead for safety.

Figure 13 shows yet another alternative implementation. In contrast to the implementation of Figure 12, this implementation uses `std:vector` for keeping all the singletons within an application. The second solution also does not loose any type information, when the pointers to singleton objects are stored. Thus it does not require any casting when the singleton pointer is accessed. This also does not have any memory overhead from the `id` class (one per templated class). However this does not address any issues about multi-threaded applications.

4 SUMMARY

The singleton pattern deals with a generic way of controlling instances of a class in an application. The traditional use of the singleton pattern can cause problems in mission-critical multi-tasking environment when the singleton pattern is present in a layer that is common to all of the tasks and each task has its exclusive dynamic

```cpp
#include <exception>

class singleton_error : public std::exception
{
public:
   virtual const char * what() const throw() {
      return "singleton error";
   }
};

template <class T>
class singleton {
public:
   static T& instance(void);
protected:
   singleton(void);
   // not requiered to be virtual at this level
   ~singleton();
private:
   static T* instance_;
   singleton(const singleton&);
   singleton& operator=(const singleton&);
};

template <class T>
inline singleton<T>::singleton(void) {
   if(instance_){
      throw singleton_error();
   }
   instance_ = static_cast<T*>(this);
}

template <class T>
inline singleton<T>::~singleton() {
   instance_ = 0;
}
```

```cpp
template <class T>
inline T& singleton<T>::instance(void) {
   if(!instance_) {
      throw singleton_error();
   }
   return *instance_;
}

template <class T>
T* singleton<T>::instance_ = 0;

// application code
// my_singleton1 is the interface for the application
class my_singleton1 :
      public singleton<my_singleton1> {
public:
   my_singleton1(void){ }
   virtual void hello_world(void) = 0;
};

// my_singleton1_impl is the actual implementation
class my_singleton1_impl : public my_singleton1 {
public:
   my_singleton1_impl(void) { }
   virtual void hello_world(void);
};

void my_singleton1_impl::hello_world(void)
{
}

int main(int /*argc*/, char* /*argv*/[])
{
   my_singleton1_impl ms1; // scoped singleton

   my_singleton1::instance().hello_world();

   return 0;
}
```

Figure 13: Singleton that does not use a vector and id for distinguishing different singletons

memory pool. In this paper we have discussed one problematic area in the context of multi-tasking programming environment and have proposed solutions that may be appropriate in some situations. We have also presented different variations of the implementation strategy and presented sample implementation details in C++. Table 1 summarizes the pros and cons of different implementation strategy. There may be different factors that govern the choice of one implementation strategy over the other.

5 ACKNOWLEDGEMENTS

The authors wish to thank Mogens Hansen for providing constructive criticisms on earlier drafts of this paper, answering many implementation related questions promptly and suggesting two alternative ways of implementing the singleton pattern that do not have any circular dependencies. Amit Bhonsle helped with the implementation of some of the ideas presented in this paper.

Implementation of	Summary
Figure 5	Simple and straightforward implementation. Can be used for multi-threaded applications. May have some performance problems if the singleton implements a publisher-subscriber
Figure 8	This approach creates one singleton for every tasks within the application. Even if any tasks is terminated, the singletons `instance` would not have any effect on the rest of the application. Maintenance of enumeration types can create problems. One of the problems is that in some cases, synchronization may need to be implemented across different tasks. The application programmer has to be careful about matching the value of the enumeration and the type used in the cast.
Figure 10	With this approach, singletons can be pre-instantiated and deallocated appropriately (order of deallocation can be specified). One of the advantages is that application does not need to check the `if` statement (faster access) whether the singleton instance is already created. However on the downside, can create circular dependencies, maintenance of the enumeration types is a potential source of errors and type information is also lost. However even if a task dies, the rest of the application would not be affected because of singleton problems.
Figure 12	No circular dependencies, no programmer maintained enumeration of types and no loss of type information. Uses vector for keeping track of a list of singletons and each singletons must be instantiated explicitly. Adds a bit of memory overhead as the class `id` needs to be instantiated for each singletons. singletons are deallocated automatically once the corresponding variable goes out of scope. Before a singleton is used, the singleton needs to be explicitly instantiated.
Figure 13	There is no need for dynamic allocation and `std::vector`. Also does not have any performance overhead from `dynamic_cast` and memory overhead from the singleton specific `id` (class member for distinguishing between different singletons. However this does does not address anything about multi-threaded error scenarios.

Table 1: Summary of different implementation strategy

REFERENCES

[Alexandrescu, 2001] Alexandrescu, A., 'Modern C++Design: Generic Programming and Design Patterns Applied', Addison-Wesley Longman, Inc., ISBN: 0201704315, 352 pages, 2001.

[Coplien, 1995] Coplien, J., Curiously Recurring Template Patterns, *C++Report*, February 1995, pp. 24–27.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley Longman, Inc., ISBN: 0201633612, 395 pages, 1995.

[Pant, 2002] Pant, Y., 'Using Rhapsody for Developing an OO Framework for CDMA RAN', Motorola Software Engineering Symposium, May 2002.

[Schmidt, 1999] Schmidt, D.C., Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components, *C++Report*, SIGS, vol. 11, no. 9, September 1999.

ABOUT THE AUTHORS

Yagna Pant is a Senior Staff Engineer at Motorola. He has a Masters degree in Software Engineering from the University New South Wales, Sydney, Australia. He works on the development of object-oriented, real-time software. His other interests are object-oriented design patterns, software reuse, software metrics, cellular systems, network management and voice over Internet Protocol.

Kazuhiro Ondo is a Senior Staff Engineer in the CDMA software development group in Global Telecom Solution Sector in Motorola Inc. He is engaged in developing CDMA call processing software with Object Oriented program on real-time embedded system. He can be reached at Kazuhiro.Ondo@motorola.com