Save Time With Software Patterns

by Muaz Niazi

Citation:
Niazi, Muaz (2000). "Save time with patterns: Singleton design pattern." Visual C++ Developers Journal **3**(5).

Use the Singleton pattern to implement a simple console application and an ATL COM object

Programmers benefit from software patterns in many ways. Patterns help solve practical software problems by documenting existing expertise; this way, developers can benefit from the experience of seasoned pros. Patterns provide a shared vocabulary for problem solving and show more than just the solution to the problem at hand—they demonstrate potential solutions to other related problems as well.

What you need:

Windows NT,
95, or 98
Visual C++ 5 or 6

Patterns solve real problems. Like court decisions, patterns convey the various decisions regarding the design and analysis of object-oriented software. By using patterns, developers can talk in terms of interactions of patterns, instead of focusing on the tiny details of classes and objects.

In this article, I'll explain what software patterns are, how they are structured, and how you use them. You'll learn to use a common software pattern, the Singleton pattern, to implement a simple console application and an ATL COM object.

Writing a pattern isn't easy (see the sidebar, "What is a Pattern?"). A pattern author has to justify the pattern by giving examples of how to use it in different domains, usually at least two. For example, the abstract factory-design pattern is documented in InterViews and ET++, and the Singleton design pattern is documented in Smalltalk-80, as well as the InterViews UI Toolkit. The author must also provide any related patterns so developers can understand their differences. If developers use the patterns together in some scenarios, the author has to document this in the related-patterns section of the pattern documentation.

Experts such as Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides categorize 23 design patterns [1, 2]. According to these experts, patterns have three basic purposes: creational, structural, and behavioral. Creational patterns deal with creating objects, whereas structural patterns deal with composing classes or objects. And behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

A pattern's scope specifies whether you can apply a pattern primarily to classes or to objects. As the name implies, class patterns deal with classes and their subclasses. Object patterns deal with relationships at run time and are in essence, dynamic. Most of the patterns are dynamic, and so belong to the object pattern category.

Considering both the purpose and scope of patterns, you can see that creational class patterns should be the ones that defer object creation to subclasses, while creational object patterns defer it to another object. Similarly, behavioral class patterns use inheritance mainly to depict algorithms and flow, whereas behavioral object patterns describe how a group of objects exhibits behavior that no single object can perform alone. UML comes to our aid by efficiently depicting the relationships between objects—which are dynamic. Finally, structural class patterns use inheritance to compose classes, while their object counterparts describe ways to assemble objects.

Design patterns help programmers find appropriate objects by identifying less obvious abstractions. In addition, they can help designers decide on the level of object granularity required. They also specify the interface (exposed part) of the objects, as well as the implementation of the objects; put reuse mechanisms such as inheritance and composition to work; and can certainly help in relating runtime and static structures. Lastly, patterns make designs adaptable to change.

Enter the Singleton Pattern

The Singleton pattern is an object-creational pattern that ensures a class has only one instance. For example, you might require classes to have a single instance if you have many printers on a system, but only one printer spooler. You can deal with this kind of problem by using global variables, which are easily accessible but don't force the class to have a single instance. The Singleton offers a better solution than global variables; it provides a global access point for the class.

Use the Singleton when the class should have only one instance accessible from a single well-known access point. In addition, the sole instance should be extensible through subclassing, and clients should be able to use it or an extended instance without modifying the code (see Figure 1). The Singleton class has a static variable with a unique instance.

The Singleton design pattern offers designers many benefits. It provides controlled access to the sole instance. As the class encapsulates the sole instance, you can control access as strictly as you want. Also, you have a reduced name space, which proves a better solution than global variables because it eliminates filling out the name space unnecessarily. The Singleton also permits you to refine operations and representation. You can subclass the Singleton and easily configure the application to use it, and even control the number of instances by changing the method of accessing the instance.

Figure 1. The Singleton's Structure
Click here.

The Singleton also permits a variable number of instances, making it easy to switch to more instances later, if you need them. Although you can do the same thing using static-member functions, the Singleton pattern is better and more flexible because in C++, static member functions are never virtual, making it difficult to change the design for multiple instances.

Figure 2. Inspect the Program Output Click here.

To learn how to use the Singleton pattern with a simple console application, create an empty Win32 console project using the Visual C++ AppWizard. Add the source files Singleton.h, Singleton.cpp, and main.cpp (download these), then populate those files with these three code snippets: header file, source file, and driver program. If you compile and run the program from a command prompt, you'll get the console output (see Figure 2). To start, look at the header file for the Singleton pattern:

```
//Singleton.h
#ifndef __SINGLETON_H__
#define __SINGLETON_H__
#include <windows.h>
class CSingleton
{
public:
   static   CSingleton* Instance();
   ULONG GetData();
   void SetData(const ULONG
         a_unlData);
protected:
   CSingleton();
   ULONG m_unlSomeData;
private:
   static CSingleton* s_snglInstance;
};

#endif //__SINGLETON_H__
```

I've included the standard windows.h header so you can use the standard Win32 datatypes in your program. This file contains the declaration of the CSingleton class, the class implementing the Singleton pattern. To illustrate that the class has no more than one instance, use the data variable m_unlSomeData. The GetData() and SetData() functions provide access to this data. The Instance() function gives access to the single instance, defined here as a static pointer to the class itself. Note that the sole constructor of the class isn't accessible from outside the class, ensuring that the only way to get the instance of the class is through the Instance() method.

Next, implement the Singleton class in Singleton.cpp (see Listing 1). In this file, you initialize the static variable. In C++, static member variables aren't attached to a particular instance; instead, you can access them without creating an instance of the class. The CSingleton class's constructor initializes the data to zero. The two functions you need for the data are simple, and set or get the values, respectively. The implementation's most important part is the Instance() function, which actually creates the one instance and gives it to this class's clients.

Finally, you get to the driver program containing the main function:

```
//Main file
```

```
#include <conio.h>
#include <iostream.h>
#include "Singleton.h"

void main()
{
   CSingleton* psnglObject1,*
         psnglObject2;
   psnglObject1 =
         CSingleton::Instance();
   psnglObject2 =
         CSingleton::Instance();
   psnglObject1->SetData(10);
   cout << "Singleton Object 1 has
         data " <<
         psnglObject1->GetData()
         << endl << flush;
   cout << "Singleton Object 2 has
         data " <<
         psnglObject2->GetData()
         << endl << flush;
   getch();
}
```

As you can see, you declare two pointers to the CSingleton class. You can't directly call the CSingleton constructor. Instead, you call the Instance() method, the only way to gain access to the instance. Now, you use the accessor SetData() function to set the object's data. However, you have the same data for both pointers! So, regardless of the number of clients, the CSingleton class can hook them on to the same instance, illustrating that the class is true to its name (see Figure 2).

Implement the Pattern in ATL

Now that you know how to implement a Singleton in C++, implement the pattern in ATL and see how it handles the task. First, take a look at how ATL implements COM (see Figure 3). Of the classes ATL uses to implement COM, CComObject<>, CComAggObject<>, and CComPolyObject<> implement the three methods of IUnknown. CComObjectRootEx<> handles object-reference count management for both nonaggregated and aggregated objects. It holds the object reference count if your object isn't being aggregated, and it holds the pointer to the outer unknown if your object is being aggregated.

Figure 3. How ATL Implements COM Click here.

For aggregated objects, you can use the CComObjectRootEx<> method to handle the failure of the inner object to construct, and you can use it to protect the outer object from deletion when inner interfaces are released or the inner object is deleted. CComObjectRootEx<> provides the default QueryInterface() mechanism based on COM map entries. It's essential for all ATL objects to have a single instance of CComObjectRoot<> or CComObjectRootEx<> in their inheritance. For ATL, CComObject<> and CComAggObject<> are the best way to deal with COM objects, without and with aggregation, respectively. However, if the macro DECLARE_POLY_AGGREGATABLE is defined, ATL uses CComPolyObject<> instead of the other two classes.

This approach supports both aggregatable and non-aggregatable objects, but it might result in a slightly larger module size than expected. However, for a full control or an Internet Explorer control, the ATL Object Wizard automatically adds the macro. You derive from IDispatchImpl<> for a default implementation of IDispatch. If you want to report errors back to the client using IErrorInfo, your object must support ISupportErrorInfo. The ATL template class ISupportErrorInfoImpl<> provides an easy way to implement this, if you have a single error-generating interface.

Now look at CComCoClass<>, used by ATL to define the default class factory and aggregation model for your object. ATL also uses the typedef mechanism to implement default behavior. For example, the DECLARE_AGGREGATABLE macro uses typedef to define a type called CreatorClass, referenced throughout ATL. CComCoClass<> specifies these two macros: DECLARE_CLASSFACTORY, which declares the class factory to be CComClassFactory<>, and DECLARE_AGGREGATABLE, which declares the object is aggregatable. Here's an actual code snippet from atlcom.h:

```
template <class T, const CLSID*
          pclsid = &CLSID_NULL>
class CComCoClass
{
public:
   DECLARE_CLASSFACTORY()
   DECLARE_AGGREGATABLE(T)
   . . .
};
```

To override any of these, you can use any other macro in your class definition. Returning to the Singleton pattern, to use the ATL's default implementation, you simply have to define the macro DECLARE_CLASSFACTORY_SINGLETON in your class, and it overrides the default behavior:

```
class CMySingleClass : . . ., public
      CComCoClass<...>
{
. . .
DECLARE_CLASSFACTORY_SINGLETON(CMySingleClass)
. . .
};
```

This means that ATL uses the class CComClassFactorySingleton<>, instead of CComCoClass<>, as the class factory. Here's the code for the DECLARE_ CLASSFACTORY_SINGLETON macro from atlcom.h:

```
. . .
#define
   DECLARE_CLASSFACTORY_SINGLETON(
   obj)
   DECLARE_CLASSFACTORY_EX(
   CComClassFactorySingleton<obj>)
. . .
```

To understand how ATL implements the Singleton pattern, look at the definition of CComClassFactorySingleton<>, again in your favorite atlcom.h (see Listing 2). The code reveals a number of things here. First, ATL has added a member of type CComObjectGlobal<> to the class CComClassFactorySingleton. Next, the default implementation of ATL ensures the Singletons aren't aggregatable. Below the comment, the debug version should give an ASSERT() failure if pUnkOuter is non-null. In general, the class doesn't support aggregation. What might not be obvious is that you'll have problems implementing Singletons in DLLs. (See Resources for a description and workaround.)

Technically, this procedure meets the formal definition of a Singleton. Every client gets a reference to the same instance of the class through a well-known access point. One problem is that CreateInstance() behaves "unnaturally" according to COM, because it's supposed to give new instances each time. Considering the norm of COM programming, clients unaware of your implementation—and, in OOP, clients are supposed to be unaware of the implementation and focus instead on the class's interface—might be surprised if your implementation expects two different objects from the CreateInstance() method.

CComObjectGlobal<> manages a reference count on the module containing your Base object. It ensures the object won't be deleted, as long as the module isn't released. The object will be removed only when the reference count on the entire module goes to zero. A single CComObjectGlobal<> object is being used—it has a member variable m_hResFinalConstruct, which has the HRESULT value from calling FinalConstruct(). ATL uses this object for the reference counting.

Another way to use Singletons in COM: Implement your own custom interface and implement it the way you want. This implementation doesn't break COM programming norms:

```
// Custom interface
// implemented by
// class object
interface IMySingletonFactory :
         IUnknown {
      HRESULT GetTheInstance([in]
         REFIID riid,
         [out, iid_is(riid)]
```

```
         void **ppv);
}
// Implementation of GetTheInstance
HRESULT
   CMyClass::GetSingleInstance(REFIID
   riid, void **ppvObject)
{
   static CMyClass s_myClass;
   return
      s_myClass.QueryInterface(riid,
      ppvObject);
}
```

In this code, you're designing a completely new interface, IMySingletonFactory, derived from IUnknown. This process doesn't limit you, as does ATL, by IClassfactory::CreateInstance(). Instead, you do things your own way. You get access to your single instance through your lone method, GetSingleInstance()—a much better way to implement Singletons than ATL's method. However, in both cases, you're forgetting one basic COM concept: COM objects are typically in separate apartments. So, you must marshal interface pointers between two apartments. If the class object returns the interface pointer directly, and you use the Singleton in multiple apartments in one process, the code violates the rules.

This means that to use in-process Singletons, you should either mark the threading model in the registry as free or else leave out the ThreadingModel attribute altogether from the registry. The point is, Singletons have limited applicability to distributed computing unless you focus on a single state, rather than a single instance. When you need a single instance without going into distribution, the Singleton pattern works well.

The pattern revolution made developers and analysts rethink their way of getting the job done. Patterns solve complex problems. In my opinion, the Singleton design pattern is a good way to start learning patterns. You can move on to more complex patterns involving multiple classes, or even multiple patterns, in the future. And as you start using even one pattern in your designs, you'll find yourself looking for other patterns as well. At the least, you'll start to observe what goes into a design—good or bad.

Muaz Niazi is an electrical engineer with post-graduate training in software development and an MCSD certification. He is a software consultant and specializes in software engineering practices, as well as developing applications in C++, COM, ATL, and MFC. Reach him at muaz.niazi@gmail.com.

References

[1] E. Gamma*, et al.*, *Design patterns: elements of reusable object-oriented software*: Pearson Education, 1994.

[2] R. E. Johnson, "Frameworks=(components+ patterns)," *Communications of the ACM,* vol. 40, pp. 39-42, 1997.