

17. Service Wrappers

17.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of programming in C++.
2. A knowledge of component interface design.

After completing this chapter you will have:

1. A knowledge of how to encapsulate operating system functions in a programmer-friendly wrapper.
2. A knowledge of the Service Wrapper design methodology.

17.2. Introduction

A service wrapper is a component that provides access to operating system services. In most cases, these services are already available to the programmer in some form, but are difficult to access or to use properly. The service wrapper is provided to simplify access to these services.

In commercial software, the service wrapper is probably the second most common type of component after the UI Widget. In the earlier versions of Visual Basic, it was difficult or impossible to access certain operating system services. In many cases, service wrappers were the only way to access these services. (An example of such a service is the Font Selection dialog.)

The quintessential example of a service wrapper is the Microsoft Common Dialogs component, which provides access to the open, save as, and color selection dialogs, in

addition to several others. Commercial service wrappers are available for accessing Internet services such as ftp, telephony services such as autodial and terminal emulation, and many other services. In this chapter we will provide a service wrapper for the INI (initialization) file services. In most of today's software, these services have been replaced with registry services. We could have provided a service wrapper for registry services. However, registry information tends to be quite sensitive to tampering, and we chose not to enhance the student's ability to damage this information.

17.3. The Methodology

The first step in the methodology is to examine the operating system service to be encapsulated to determine if the existing interface is adequate. In most cases, the features that are to be encapsulated are already accessible to the host program in some fashion, and it is important to determine whether a component-level wrapper will be an improvement. If the existing access methods require the use of callback functions, or function calls with a large number of parameters, then a component-level interface will probably be an improvement.

The component should provide all required callback functions, and should reduce the number of function parameters, either by supplying standard defaults, or by using properties to set the value of parameters that are used repetitively. The component should provide events to report any asynchronous conditions that occur.

Where possible, functions should be reduced to variables. A function that supplies data to the operating system can usually be reduced to a variable. (Recall that a variable can be viewed as a pair of Get/Set functions).

Some pass-through functions may be required. Pass-through functions are methods that merely pass a set of arguments through to an operating system function, without changing the list or adding to it. (If a service wrapper consists only of pass-through functions, it is probably not worth the effort.)

17.4. The INI File Manager

Figure 17.1 shows the format of a typical INI file. The file is broken into sections each one of which is preceded by a string in square brackets. The name in the square brackets is known as the *Application Name*. Each section consists of a series of name-value pairs of the form MEMBERS=MEMBERS.MDB. The string to the left of the equal sign is known as the *Parameter Name*, while the string to the right is known as the *Parameter Value*. To locate a particular parameter value, it is necessary to specify the file name, the application name, and the parameter name.

```
[DBDATA]
DATADIR=E:\TESTDB
MEMBERS=MEMBERS.MDB
FINANCE=FINANCE.MDB
PRINTDIR=E:\CHURCHDB

[ProgLoc]
Household=E:\CHURCHDB\HOUSHOLD.EXE
People=E:\CHURCHDB\PEOPLE.EXE
RecordAttend=E:\CHURCHDB\RECATTND.EXE
ViewAttend=E:\CHURCHDB\CHKATTND.EXE
```

Figure 17.1. A Typical INI File.

The Windows operating system provides four functions for accessing INI files. Although the programmer may call these functions directly, we can simplify the access by encapsulating the functionality of these functions into a component. We will provide persistent properties for the file name, application name, and the parameter name.

Assuming that these properties have been assigned values, reading and writing the parameter value can be reduced to a simple property access as illustrated in Figure 17.2.

```
' Write parameter value  
INIWrapper.Value = "New Parameter Value"  
' Read parameter value  
MyString = INIWrapper.Value
```

Figure 17.2. INI File Access.

The four operating system functions are *GetProfileString*, *GetPrivateProfileString*, *WriteProfileString*, and *WritePrivateProfileString*. Figure 17.3 and Figure 17.4 show the parameters of these functions, and their usage. *GetProfileString* and *WriteProfileString* are used to access the WIN.INI file, while *GetPrivateProfileString* and *WritePrivateProfileString* are used to access other INI files. The functions *GetProfileString* and *GetPrivateProfileString* have two forms. If the parameter name is the constant NULL, the function returns a list of all parameter names in the section. If the parameter name is not NULL, the function returns the parameter value. If a list of parameter names is returned, each parameter name is terminated by the null character (\0), and the list is terminated by two consecutive null characters. The *WriteProfileString* and *WritePrivateProfileString* functions have three different functions. If both the parameter name and the value parameters are specified as NULL, the section specified by the application name is deleted from the INI file. If a parameter name is specified with a NULL constant for the value, then the name-value pair for the parameter is deleted from the section.

GetProfileString (Application,Parameter,Default,Buffer,Size)	
GetPrivateProfileString (Application,Parameter,Default,Buffer,Size,FileName)	
Application	A null-terminated string containing the application name
Parameter	The constant NULL or a null terminated string containing the parameter name
Default	A null-terminated string that will be returned if the parameter does not exist
Buffer	A pointer to a storage area where the parameter value or name list will be stored
Size	The size of the storage area pointed to by <i>Buffer</i>
FileName	The name of the INI file to be read. If a full path name is not supplied, the file is assumed to be in the <i>Windows</i> directory

Figure 17.3. GetProfileString and GetPrivateProfileString Parameters.

WriteProfileString (Application,Parameter,Value)	
WritePrivateProfileString (Application,Parameter,Value,FileName)	
Application	A null-terminated string containing the application name
Parameter	The constant NULL or a null-terminated string containing the parameter name
Value	The constant NULL or a null-terminated string containing the parameter value
FileName	The name of the INI file to be read. If a full path name is not supplied, the file is assumed to be in the <i>Windows</i> directory

Figure 17.4. WriteProfileString and WritePrivateProfileString Parameters.

Our INI Service Wrapper will allow the user to perform all INI file operations, including deletes, by assigning values to properties. The properties of the INI file Service Wrapper are listed in Figure 17.5.

Property Design Table		
Name	Type	Function
FileName	String Default=Empty	Name of the INI file. If this property contains the empty string, operations will be performed on WIN.INI.
Application	String Default=Empty	The name of the INI file section.
Parameter	String Default=Empty	The parameter name. If this string is empty, the Value property will return a list of parameter names in the section specified by <i>Application</i> .
Value	String Default=value of Default property	Returns, or sets the value of the parameter specified by <i>Application</i> and <i>Parameter</i> .
Default	String Default=Empty	Default value for parameters that do not exist.
DeleteApp	String Default=None	The name of the section to be deleted from the INI file. Restrictions: write only, run time only
DeleteParm	String Default=None	The name of the parameter to be deleted from the section specified by <i>Application</i> . Restrictions: write only, run time only

Figure 17.5. The INI Service Wrapper Property Design Table.

Most of the functionality of the INI Service Wrapper is contained in the implementation of the *Value* property. The *read* portion of the implementation is given in Figure 17.6. If no application name is specified, this function returns the value of the *Default* property. If no file name is specified, the *GetProfileString* function is used to access the WIN.INI file. Otherwise, the *GetPrivateProfileString* function is used to access the specified file. If no parameter name is specified, the *GetProfileString* or *GetPrivateProfileString* function is called with a NULL parameter name to obtain the list of parameter names in the section specified by the *Application* property. If either the application name or the parameter name does not exist in the file, the value of the *Default* property is returned. Returned strings are limited to 1024 characters.

```

BSTR CIniManagerCtrl::GetValue()
{
    CString strResult;
    static char StringBuffer[1024];

    if (m_application.IsEmpty()) // no application, return default
    {
        strResult = m_default;
    }
    else
    {
        if (m_fileName.IsEmpty()) // no file name, use WIN.INI
        {
            if (m_parameter.IsEmpty()) // no parameter, return name list
            {
                GetProfileString(m_application, NULL, m_default,
                                StringBuffer, 1024);
                strResult = StringBuffer;
            }
            else
            {
                GetProfileString(m_application, m_parameter, m_default,
                                StringBuffer, 1024);
                strResult = StringBuffer;
            }
        }
        else // file name has been specified, use it
        {
            if (m_parameter.IsEmpty()) // no parameter, return name list
            {
                GetPrivateProfileString(m_application, NULL, m_default,
                                       StringBuffer, 1024, m_fileName);
                strResult = StringBuffer;
            }
            else
            {
                GetPrivateProfileString(m_application, m_parameter,
                                       m_default, StringBuffer, 1024, m_fileName);
                strResult = StringBuffer;
            }
        }
    }
    // convert CString to BSTR
    return strResult.AllocSysString();
}

```

Figure 17.6. The Read Portion of the Value Property.

The *write* portion of the *Value* property, which is given in Figure 17.7, is simpler than the *read* portion. Before assigning a value to the *Value* property, it is necessary to assign values to the *Application* and *Parameter* properties. If either of these is empty, no action

will be taken. If no *FileName* is specified, the *WriteProfileString* function will be used to access the WIN.INI file, otherwise the *WritePrivateProfileString* function will be used.

```
void CIniManagerCtrl::SetValue(LPCTSTR lpszNewValue)
{
    // Both application name and parameter name must be specified
    // deletes are not permitted here.
    if (!m_application.IsEmpty() && !m_parameter.IsEmpty())
    {
        if (m_fileName.IsEmpty()) // no file name, use WIN.INI
        {
            WriteProfileString(m_application,m_parameter,lpszNewValue);
        }
        else // file name exists, use it
        {
            WritePrivateProfileString(m_application,m_parameter,
                                     lpszNewValue,m_fileName);
        }
    }
}
```

Figure 17.7. The Write Portion of the Value Property.

Finally, we present the implementations of the *DeleteApp* and *DeleteParm* properties in Figure 17.8 and Figure 17.9. Assigning a value to the *DeleteApp* property causes the section specified by the new value to be deleted. Assigning a value to the *DeleteParm* property causes the specified parameter to be deleted from the section specified by the *Application* property. If the application property has not been set, assigning a value to *DeleteParm* will do nothing.

```
void CIniManagerCtrl::SetDeleteApp(LPCTSTR lpszNewValue)
{
    if (m_fileName.IsEmpty()) // no file name, use WIN.INI
    {
        WriteProfileString(lpszNewValue, NULL, NULL);
    }
    else // file name exists, use it
    {
        WritePrivateProfileString(lpszNewValue, NULL, NULL, m_fileName);
    }
}
```

Figure 17.8. The DeleteApp Property.

```
void CIniManagerCtrl::SetDeleteParm(LPCTSTR lpszNewValue)
{
    // application name must be specified or nothing happens
    if (!m_application.IsEmpty())
    {
        if (m_fileName.IsEmpty()) // no file name, use WIN.INI
        {
            WriteProfileString(m_application, lpszNewValue, NULL);
        }
        else // File name exists, use it
        {
            WritePrivateProfileString(m_application, lpszNewValue, NULL,
                                     m_fileName);
        }
    }
}
```

Figure 17.9. The DeleteParm Property.

17.5. A Review of the Methodology

Because a service wrapper is designed to enhance the programmer's access to something that is already available, it is first necessary to determine whether a component is needed for a particular service. It is necessary to first list those features of the service that will be encapsulated. For each feature, describe the user access both with and without the component. Make sure to include any required header file access, initialization steps, and termination operations for each case, and also make sure to consider the case where a particular feature is used several times. If, on the whole, the access is simpler using the component, then a component is needed.

The most important simplification that a service wrapper can provide is reducing a function to a variable. In the case of the INI manager, the functions used to access a parameter are reduced to accessing the *Value* property. The access functions become transparent, and the user appears to be storing and retrieving the parameter value directly from the INI file.

The next most important simplification is reducing the number of required function parameters. This is especially useful if a control block is required for the service. Examples of services that require control blocks are listing files in a directory, and obtaining the current time. The required control blocks can be embedded in the component and made invisible to the user. Some services require call-back functions to be supplied as function parameters. Embedding these callbacks in the component is an important simplification. (Some languages do not support call backs, and cannot give access to the service without a component to provide the callback functions.)

The least important simplification is providing straight-through access to service functions. The number of parameters is not reduced, and the parameters do not undergo transformation before being passed to the service function. Despite appearances, this is still a simplification, because components do not require header files.

17.6. Conclusion

As mentioned above, the service wrapper is probably the second most commonly encountered component after the UI Widget. This is because accessing these services in Visual Basic, the most widely used language for component programming, was difficult or impossible in the earlier versions. Many services that were already accessible in Visual Basic, such as common dialog access, were encapsulated to speed the development of

programs using them. Service wrappers can be provided to specialize service access to specific needs. For example, a company may have a certain location in the registry where all initialization parameters are stored. A component could be used to simplify access to that portion of the registry, without giving access to the entire registry. Because operating system services are always expanding, there is still a great deal of development going on in this area.