

11.Accessors

11.1. 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 the Accessor-Component design methodology..

11.2. Introduction

The Accessor component gives programmatic access to an object that is either inaccessible, or accessible only through great difficulty in the base language. An example of such an object is an HTML document. Even though an HTML document is nothing more than a text file, it has a highly complex structure. It is possible to parse an HTML file in virtually any language, but it is a daunting task. There are Accessor components that permit one to download an HTML page, and analyze its structure automatically. One can enumerate tags, links, sections and perform other complex operations. These things could also be done through brute-force programming, but it is neither an easy nor a pleasant task to do so.

Another type of accessor is one that is designed to give access to a semi-persistent object that has been created by another component. Such an object may or may not be accessible in the base language. Even if the object is accessible, the Accessor can provide additional functions that simplify certain types of access.

In this chapter we will present two examples of Accessor components, an object Accessor and a file Accessor. Object Accessor functions are reasonably straightforward, and are often incorporated into another component, such as an Editor or a Viewer.

11.3. The Methodology

The two main parts of an Accessor are data elements and enumerators. Individual data items such as integers, floating point numbers, and strings, are presented through a set of data access properties. An accessor for an object should have a property for each simple data member of the object. For data elements that are arrays of simple objects, the preferred method is to use an array property, but it is also possible to use an enumerator. Sub-objects are normally associated with an enumerator, but if there is a single instance of the sub-object that is always present, then simple properties can be used as though the data elements of the sub-object belonged to the parent object.

Enumerators must be used for arrays and linked lists of sub-objects. There are two styles of enumerators, the parallel-array enumerator and the First/Next enumerator. In the parallel-array enumerator, the data members of an object are treated like a set of parallel arrays, one array for each data member. The component should be supplied with a Count property that gives the size of the parallel arrays. This sort of access gives maximum accessibility, but can be extremely slow for linked lists.

The First/Next enumerator is implemented with two methods one that initializes the enumeration to the first element of the list or array, and one that advances the enumeration to the next element. A set of simple (non-array) properties are used to access the data members of the current object. The content of these properties changes when the First and Next functions are executed. Some mechanism must be provided for signaling

the end of the enumeration. The best way is to provide the First and Next functions with a return code that indicates whether all items have been exhausted. Other methods are to provide a **Done** property or an **EndOfList** event. The following sections illustrate this methodology with two examples.

11.4. The Simple Graphical Editor Accessor

In our first example, we will present an Accessor for the Simple Graphical Editor internal object. We will limit our accessor functions to the list of graphical objects, but we could easily expand the functionality of this component to include the selection and the undo/redo lists. Before proceeding with the development of the Accessor component, it is necessary to add some additional functionality to the **CGraphicList** model. Because the Accessor must provide the type and parameters of each graphical object, it is necessary to enhance each object with functions to provide this information. Furthermore, it is necessary to enhance the **CGraphicList** with functions that give external access to the list of graphical objects. Due to the simplicity of our requirements, the only new function required by the **CGraphicList** object is *GetHead*, which will return the first element in the list of graphical objects. Each of the classes **CGraphicObject**, **CCircle**, and **CRectangle** must have three new virtual functions, *GetType*, *GetParms*, and *GetParmCount*. Rather than return parameters by name, each object will create an array of long integers and place all of its parameters into the list. This is done to avoid over-complicating the object structure and the Accessor component interface. The **CGraphicObject** functions are pure virtual. For **CCircle** and **CRectangle**, the *GetParmCount* and *GetType* functions return constants that are appropriate for the object.

The *GetParms* function is slightly more complicated. Figure 11.1 illustrates the *GetParms* function for **CRectangle**, that of **CCircle** is similar.

```
long * CRectangle::GetParms()  
{  
    // The Accessor provides lists of parameters for each object  
    // It does not name them individually  
    long * rv = new long[4];  
    rv[0] = Left;  
    rv[1] = Top;  
    rv[2] = Width;  
    rv[3] = Height;  
    return rv;  
}
```

Figure 11.1. The CRectangle GetParms Function.

It would have been possible to include line and fill colors in the list of parameters created by the *GetParms* routine, but since these are properties of all objects, it is more straightforward to provide explicit accessor properties for them.

The design of the SGE Accessor follows a pattern that is similar for all Accessors. A set of methods is provided to iterate through a list of objects, and a set of properties is provided to access the object members. Because we do not wish to provide editing capabilities in the Accessor, all properties are read-only. It is a common practice to combine accessor functionality with that of a Background Editor to provide both access and editing capability. Figure 11.2 gives the description of the SGE Accessor methods.

Method Description			
Name	FirstObject		
Return Value	Long Integer		
Description	Begins the iteration process with the first element of the list of graphical objects. If the list is empty, this function returns 0 otherwise it returns 1. If this function returns zero, then the Accessor properties have undefined values, otherwise they return the properties of the first object in the list.		
Arguments	Name	Type	Description
Void			

Method Description			
Name	NextObject		
Return Value	Long Integer		
Description	Begins the iteration process with the first element of the list of graphical objects. If there is no next object, this function returns 0 otherwise it returns 1. If this function returns zero, then the Accessor properties have undefined values, otherwise they return the properties of the first object in the list. There are several conditions under which there is no next object: no internal object model has been supplied, the FirstObject function has never been called, there is no current object, and there is a current object, but it is the last object in the list.		
Arguments	Name	Type	Description
Void			

Figure 11.2. The FirstObject and NextObject Method Descriptions.

The methods described in Figure 11.2 are typical of those used to traverse a linked list of objects. The *FirstObject* function starts the iteration while the *NextObject* continues the iteration. Iterator methods generally occur in pairs in this fashion. The iterator variable is contained in the component itself. Other function pairs can be used to provide other types of access. For example, a pair of reverse-order functions could be provided, or a pair of functions for accessing only circles could be provided. In some cases a single *NextObject* method may be provided, with the *FirstObject* method determining the subsequent operation of the *NextObject* function. This technique produces fewer

programming errors, but permits only one iteration to be performed at a time. The

FirstObject function is given in Figure 11.3.

```
long CSGEAccessorCtrl::FirstObject()
{
    // returns zero if end of list found, one otherwise
    if (Model == NULL)
    {
        CurrentObject = NULL;
        // End of list found
        return 0;
    }
    CurrentObject = Model->GetHead();
    if (CurrentObject == NULL)
    {
        // List is Empty
        return 0;
    }
    else
    {
        if (Parms != NULL)
        {
            delete [] Parms; // clean up any previous access
        }
        // Get object parameters, parm count, and type
        ParmCount = CurrentObject->GetParmCount();
        Parms = CurrentObject->GetParms();
        Type = CurrentObject->GetType();
        // object is usable
        return 1;
    }
}
```

Figure 11.3. The FirstObject Method Implementation.

Figure 11.4 shows the implementation of the *NextObject* method.

```

long CSGEAccessorCtrl::NextObject()
{
    if (Model == NULL || CurrentObject == NULL)
    {
        return 0;
    }
    // go to next object in list
    CurrentObject = CurrentObject->Next;
    if (CurrentObject == NULL)
    {
        // clean up if end of list is found
        if (Parms != NULL)
        {
            delete [] Parms;
            Parms = NULL;
        }
        ParmCount = 0;
        Type = 0;
        // end of list found
        return 0;
    }
    else
    {
        // extract object type, parm count and parameters
        ParmCount = CurrentObject->GetParmCount();
        Parms = CurrentObject->GetParms();
        Type = CurrentObject->GetType();
        // object is usable
        return 1;
    }
}

```

Figure 11.4. The NextObject Implementation.

Figure 11.5 gives the property design table for the SGE Accessor component. Since the design of these properties is straightforward, we will not discuss them individually. The properties give us the number of objects, the number of parameters for each object, the values of the parameters, and individual access to the global properties of each object. Default values are returned if there is no current model or no current object.

Property Design Table		
Name	Type	Function
ModelHandle	Long Integer Maximum=N/A Minimum=N/A Default=0	Provides the address of the modeling object cast to a long integer. NULL values are ignored. Restrictions: Run Time Only
ItemCount	Long Integer Maximum=N/A Minimum=0 Default=0	A count of the number of graphic objects in the current model. Zero if there is no model Restrictions: Read Only
Type	Long Integer Maximum=2 Minimum=0 Default=0	A Numeric value indicating the type of the current object. 0=No Current Object 1=Rectangle 2=Circle Restrictions: Read Only
Parm	Long Integer Array Maximum=N/A Minimum=N/A Default=NONE	An array of properties for the current object. If there is no model or no current object, then this array has zero size. Size is given by the ParmCount property. Restrictions: Read Only
ParmCount	Long Integer Maximum=N/A Minimum=0 Default=0	The number of elements in the Parm array. Zero if there is no model or no current object. Restrictions: Read Only
ObjectHandle	Long Integer Maximum=N/A Minimum=N/A Default=0	The address of the current object cast to a long integer. NULL if there is no current object or no model. Restrictions: Read Only
FillColor	Color Default=Black	The FillColor of the current object. Black if no current object or no model. Restrictions: Read Only
LineColor	Color Default=Black	The LineColor of the current object. Black if no current object or no model. Restrictions: Read Only
Selected	Boolean Default=FALSE	Returns TRUE if there is a model, a current object, and the object is selected. Returns FALSE otherwise. Restrictions: Read Only

Figure 11.5. Properties of the SGE Accessor.

The implementations for the properties listed in Figure 11.5 are straightforward. The only unusual feature is the *Parm* property, which is an array of integers. When the *Parm*

property was declared, it was declared with a single parameter named *ParmNumber*. The implementation of the property uses this parameter to look up the property value in the property array. Figure 11.6 shows the implementation of the *GetParm* function for the property array, and the implementation of the *GetSelected* function for the *Selected* property for contrast.

```
// read-only property array
long CSGEAccessorCtrl::GetParm(long ParmNumber)
{
    // property array must exist, and ParmNumber must be legal
    if (Parms != NULL && ParmNumber > -1 && ParmNumber < ParmCount)
    {
        return Parms[ParmNumber];
    }
    return 0;
}

// read-only property
BOOL CSGEAccessorCtrl::GetSelected()
{
    if (CurrentObject != NULL)
    {
        // extraction not required: part of all graphic objects
        return CurrentObject->Selected;
    }
    return FALSE;
}
```

Figure 11.6. The GetParm and GetSelected Property Implementations.

11.5. A Text-File Accessor

The Text File Accessor component will allow a text file to be treated as an array of lines. The component has three properties, one to supply the file name, one to give the number of lines in the file, and one that will act as the array of lines. The property design table is given in Figure 11.7.

Property Design Table		
Name	Type	Function
InputFile	String Default=Empty	Provides the name of the text file to be indexed. Restrictions: Run Time Only
LineCount	Long Integer Maximum=N/A Minimum=0 Default=0	A count of the number of lines in the current file. Zero if there is no file has been supplied. Restrictions: Read Only
Line	String Array Default=Empty Str.	An array of lines for the current file. If there is no current file, then this array has zero size. Size is given by the <i>LineCount</i> property. Restrictions: Read Only

Figure 11.7. The Text File Accessor Property Design Table.

For the implementation of the Text File Accessor, we will make use of the **CLineList** class developed in Chapter 10 for the Text File Viewer. When a new file name is supplied by assigning a value to the *InputFile* property, the file is opened, it is completely read, and an array is created that gives the starting position and length for each line in the file. These actions are identical to those taken for the Text File Viewer. The reader should see Chapter 10 for the details.

When the property *Line* is accessed, the appropriate line is read from the file, using the data accumulated when the value was initially assigned to the *InputFile* property. An example of the usage of this component is given in Figure 11.8. This Visual Basic subroutine opens a text file, retrieves the total number of lines in the file, and then retrieves the text of three selected lines.

```
Private Sub Command1_Click()  
    Dim S1 As String, S2 As String, S3 As String  
    Dim LineCount As Integer  
    File1.InputFile = "tarzn10.txt"  
    LineCount = File1.LineCount  
    S1 = File1.Line(35)  
    S2 = File1.Line(457)  
    S3 = File1.Line(228)  
    ...  
End Sub
```

Figure 11.8. Text File Accessor Usage.

11.6. A Review of the Methodology

The preceding sections have illustrated the design methodology for Accessor components. The methodology is different depending on whether you are designing an object accessor or a file accessor. For an object accessor, it is necessary to start with a complete description of the object to be accessed. The first step is to separate the fixed parts of the object from the repeating parts. Each part must be further separated into items with simple and complex types. (An item with a complex type is either a structure or another object.) A read-only property is created for each item with a simple type.

If a repeating item is an array of a simple type, then a read-only array property is provided for the item. If the repeating item is a linked list, or an array of complex items, then iterator methods must be provided to select a single item from the repeating group. If the repeating item is an array, a read-only selection property can be used instead of the iterator methods. (A selection property sets the current index for the array.)

This process is repeated in a recursive fashion for items with complex types. For polymorphic types, two different approaches are possible. It is possible to supply accessor methods and properties for each of the different derived types. In most cases, this will be the preferable approach. Where it is possible, the unique parameters of each

object can be gathered into an array, but this can be done only if all items have the same type. In this respect the SGE Accessor is the exception rather than the rule. As is to be expected, providing accessor properties and methods for each derived type can be quite cumbersome.

A list of exception conditions should be developed during the process of creating the accessor properties and methods. For each exception condition, the technique for reporting the condition must be documented. In some cases, the exception condition will be reported using the return value of a method. In other cases it will be reported by firing an event. Events must be designed for each condition that is to be reported. If the user program is expected to take an action that is unique to the particular exception condition, then the condition should have its own event, otherwise it is possible to group several conditions together into a single event. In any case, the parameters of the event should give information about how and why the exception condition occurred.

11.7. Conclusion

Accessors can simplify programs that must deal with complicated objects or file structures. A significant portion of the code required to access parts of objects or files can be hidden within the Accessor component, allowing the programmer to deal with simple properties and methods. Although independent Accessors are common, it is equally common to see Accessor functionality added to other types of components.

11.8. Exercises

1. Integrate the SGE Accessor features into the Simple Graphical Editor and the SGE Background Editor.

2. Pick your favorite example from this book (other than those found in this chapter), and add accessor features to it.
3. Design an accessor for some object that you designed as part of a project for some other course.
4. Design an accessor for BMP files. See the Visual C++ documentation for the format of these files.