

12.Caches

12.1. 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 enhance a program's storage facilities with Cache components.
2. A knowledge of the Cache-Component design methodology.

12.2. Introduction

In its simplest form, a cache is simply a place to store data. In that respect it provides little more functionality than a variable. Caches become useful and interesting when they are equipped with additional capabilities. Regardless of the internal complexity of a cache, it can generally be used as if it were a simple variable. This allows the Cache designer to enhance the host language (Visual Basic or whatever) with “variables” that provide new powerful capabilities. In this chapter, we will give several examples, and then elaborate the design principles that were used to create the examples.

12.3. The Methodology

The process of designing a Cache component consists of four steps, specifying the input view or input format of the data, specifying the output view or output format of the data, designing the internal storage mechanisms, and specifying the access methods for

the output. If the Cache is simply a data repository, the input and output view will be the same. However, if the Cache transforms the data in some way, then the two may be quite different. Even a simple sorter provides different views of its input data. The input is a collection of unrelated data items, while the output is an ordered list of items. Even more substantial transformations are possible.

The internal storage problem can be quite complicated. The most popular Cache component is the Visual Basic Database component that uses a relational database for data storage. The Cache components that we will deal with in this chapter use much simpler data storage techniques. The access methods determine how data is returned to the containing program. Many different techniques are possible. One can use the various enumerators described in Chapter 11, or one can use a stacklike “pop” access through a single variable. Figure 12.1 shows the relationship between the various parts of the design. The remainder of this chapter illustrates Cache design principles through several examples.

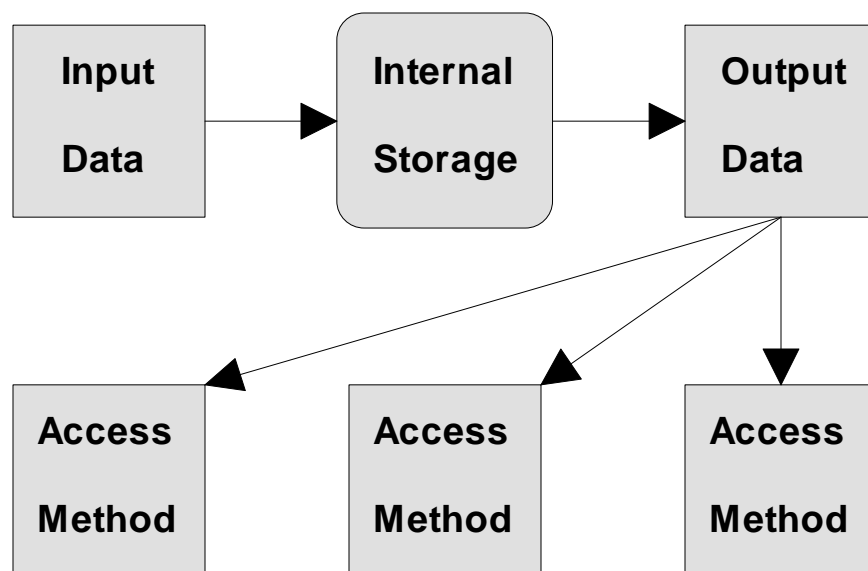


Figure 12.1. Cache-Component Structure.

12.4. Two Simple Caches

The simplest non-trivial data structures that we deal with in elementary Computer Science are stacks and queues. Using components to implement stacks and queues can greatly simplify the programs that use them. Figure 12.2 shows how stack and queue components would be used in a Visual Basic program.

```
Stack1.Value = 1
Stack1.Value = 2
Stack1.Value = 3
' Messages in Three Two One order
MsgBox Stack1.Value
MsgBox Stack1.Value
MsgBox Stack1.Value
Queue1.Value = 1
Queue1.Value = 2
Queue1.Value = 3
' Messages in One Two Three order
MsgBox Queue1.Value
MsgBox Queue1.Value
MsgBox Queue1.Value
```

Figure 12.2. Using Stack and Queue Caches.

In Figure 12.2, the stack Cache acts as if it were a variable named “Stack1.Value.” Pushes and pops are done by assigning values to the variable or by accessing the variable. If we need to look at the top of the stack without popping, we could add a *Peek* property to the component, and permit references of the form “Stack1.Peek” or “Stack1.Peek(10)”.

The unfortunate drawback of building such components is that a different component is required for each different data type. One could solve this problem by using a dynamic type such as the Visual Basic Variant, or by using generic pointers. These techniques are beyond the scope of this chapter.

The first step in designing the integer stack and queue components is to create an objects to implement the data structures. In both cases, we will use a linked list to hold the data items. Figure 12.3 shows the definition of the list element. The list elements will be managed by the **CHolderList** object, which is declared to be a friend of the list-element object. **CHolderList** will have two different implementations, one for the stack component, and one for the queue component. We will present the queue implementation, and then describe the differences between the queue and the stack. The declaration of the **CHolderList** class is given in Figure 12.4.

```
class CHolderList;
class CHolder
{
    friend CHolderList;
protected:
    CHolder * Next;
    long Value;
public:
    CHolder();
    CHolder(long NewValue);
    virtual ~CHolder();
};
```

Figure 12.3. The CHolder Class.

```
class CHolderList
{
protected:
    CHolder * Tail;
    CHolder * Head;
    long Count;
public:
    // List management functions
    void Clear(void);
    long GetCount(void);
    BOOL IsEmpty(void);
    long PopValue(void);
    void AddValue(long NewValue); // Insert into queue
    // Constructor and Destructor
    CHolderList();
    virtual ~CHolderList();
};
```

Figure 12.4. The CHolderList Class.

The primary queue management is done with the *AddValue* and *PopValue* functions. *AddValue* inserts a new integer into the queue, while *PopValue* removes an integer and returns its value. These two functions are given in Figure 12.5.

```
void CHolderList::AddValue(long NewValue)
{
    CHolder * Temp = new CHolder(NewValue);
    if (Head == NULL)
    {
        Head = Temp;
    }
    else
    {
        Tail->Next = Temp;
    }
    Tail = Temp;
    Count++;
}

long CHolderList::PopValue()
{
    if (Head == NULL)
    {
        return 0;
    }
    long rv = Head->Value;
    CHolder * Temp = Head;
    Head = Head->Next;
    delete Temp;
    Count--;
    if (Head == NULL)
    {
        Tail = NULL;
    }
    return rv;
}
```

Figure 12.5. The AddValue and PopValue Functions.

The **CHolderList** class has three additional functions, *IsEmpty* which can be used to determine whether there are items in the queue, *GetCount* which returns the number of items in the queue, and *Clear* which empties the queue. These functions are given in Figure 12.6.

```

BOOL CHolderList::IsEmpty()
{
    if (Head == NULL)
    {
        return TRUE;
    }
    return FALSE;
}

long CHolderList::GetCount()
{
    return Count;
}

void CHolderList::Clear()
{
    while (Head != NULL)
    {
        CHolder * Temp = Head;
        Head = Head->Next;
        delete Temp;
    }
    Count = 0;
    Tail = 0;
}

```

Figure 12.6. The IsEmpty, GetCount, and Clear Functions.

In the queue component, the *AddValue* and *PopValue* functions are encapsulated in the Get and Set routines for the *Value* property. The component has two read-only properties that give access to the *GetCount* and *IsEmpty* functions, and a method that gives access to the *Clear* function. Figure 12.7 gives the property design table for the queue component, while Figure 12.8 contains the method description of the *Clear* method.

Property Design Table		
Name	Type	Function
Value	Long Integer Default=Empty	Inserts values into and extracts values from the queue. Restrictions: Run Time Only
Count	Long Integer Maximum=N/A Minimum=0 Default=0	Returns the number of items currently in the queue. Restrictions: Read Only
Empty	Boolean Default=Empty Str.	Returns True if the queue is empty, False otherwise Restrictions: Read Only

Figure 12.7. The Queue Property Design Table.

Method Description			
Name	Clear		
Return Value	Void		
Description	Deletes all elements from the queue.		
Arguments	Name	Type	Description
Void			

Figure 12.8. The Queue Clear Method.

The Stack component is identical to the queue component in all respects except one, namely, the implementation of the *AddValue* function of the **CHolderList** class. Figure 12.9 shows the implementation of this function. Although no other changes are required, the *Tail* element of the **CHolderList** class could be eliminated for efficiency.

```

void CHolderList::AddValue(long NewValue)
{
    CHolder * Temp = new CHolder(NewValue);
    Temp->Next = Head;
    Head = Temp;
    Count++;
}

```

Figure 12.9. The Stack AddValue Function.

12.5. The Randomizer Cache

Although stack and queue components can be used to conveniently store large quantities of data, we can create more interesting examples by enhancing their internal structure. For example, suppose we modify the Queue component so that it returns values in random order. We could use such a component to deal bridge hands, as illustrated in Figure 12.10. (See the Cards control of Chapter 14 for an example of how the output of the randomizer might be used.)

```
Dim West(13) As Integer, East(13) As Integer
Dim South(13) As Integer, North(13) As Integer
For I = 1 To 52
    Randomizer1.Value = I
Next I
For I = 1 To 13
    West(I) = Randomizer1.Value
    East(I) = Randomizer1.Value
    North(I) = Randomizer1.Value
    South(I) = Randomizer1.Value
Next I
```

Figure 12.10. Dealing Bridge Hands.

In Figure 12.10, each card in the deck is numbered from 1 to 52. These values are inserted into the Randomizer Cache, and then placed in four different hands. Because the Randomizer Cache returns values in random order, each hand will be dealt at random.

To create the Randomizer Cache, we start with the **CHolder** and **CHolderList** classes used by the Stack and Queue components. The **CHolder** class will not be modified, but we will make the appropriate changes to the **CHolderList** class to perform the randomization. The external interface of the Randomizer Cache is identical to that of the Stack and Queue components. The changes to the **CHolderList** class definition are

given in Figure 12.11. A new data item, *Dirty* has been added along with a new internal utility function, *Randomize*.

```
class CHolderList
{
protected:
    BOOL Dirty;
    CHolder * Tail;
    CHolder * Head;
    long Count;
    void Randomize(void);
public:
    void Clear(void);
    long GetCount(void);
    BOOL IsEmpty(void);
    long PopValue(void);
    void AddValue(long NewValue);
    CHolderList();
    virtual ~CHolderList();
};
```

Figure 12.11. The Randomizer CHolderList Class.

The two functions of the **CHolderList** class, *PopValue* and *AddValue*, have been modified to perform the randomization. The data item *Dirty* will be initialized to FALSE by the **CHolderList** constructor. When *AddValue* is called, it sets the *Dirty* item to TRUE. When *PopValue* is called, it tests the value of *Dirty*, and if it is TRUE, then it calls the *Randomize* function. The *Randomize* function will randomize the order of the list, and set *Dirty* to FALSE. The only significant change to the implementation of the **CHolderList** class is the *Randomize* function, which is given in Figure 12.12.

```

void CHolderList::Randomize()
{
    if (Count <= 1) // nothing to do
    {
        return;
    }
    // Create the sorting arrays
    long * Array = new long [Count];
    long * Random = new long [Count];
    long i = 0;
    for (CHolder * Temp = Head ; Temp != NULL ; Temp=Temp->Next, i++)
    {
        Array[i] = Temp->Value;
        Random[i] = rand();
    }
    // Sort both arrays using insertion sort
    for (i=1 ; i<Count ; i++)
    {
        long j = i-1;
        long x = Array[i];
        long y = Random[i];
        for ( ; j>=0 && Random[j]>y ; j--)
        {
            Array[j+1] = Array[j];
            Random[j+1] = Random[j];
        }
        j++;
        Array[j] = x;
        Random[j] = y;
    }
    // insert values back into the linked list
    for (Temp = Head, i = 0 ; Temp != NULL ; Temp=Temp->Next, i++)
    {
        Temp->Value = Array[i];
    }
    // destroy the two arrays
    delete [] Array;
    delete [] Random;
}

```

Figure 12.12. The Randomize Function.

The Randomize function creates two arrays, *Array*, to hold the values from the linked list, and *Random* to hold a random number. Once the values of these arrays have been filled in, the two arrays are sorted in parallel using the value of *Random* as the key. Once the sort is complete, the values from *Array* are inserted back into the linked list, and the two arrays are destroyed.

12.6. The Word Extractor Cache

Although in most cases, the data retrieved from a cache will be identical to the data inserted into the cache, useful components can be created by adopting two different views of the data, one for input and one for output. For caches of this type, there is some overlap in functionality between the Cache component and a Filter component (See Chapter 13), but the function of a Cache component is more restricted. As an example, let us consider the Word-Extractor Cache. The input to this cache will be a line of text, however the data will be accessed as an array of words rather than as a line of text.

To parse the words of the line of text, we will use a technique similar to the one that we used in the text file Accessor (see Chapter 11). For each word in the line, we will keep track of the position and the length of the word using the **CWordLink** class of Figure 12.13.

```
class CWordLink
{
public:
    CWordLink * Next;
    long Position;
    long Length;
    // Construction and Destruction
    CWordLink(long NewPos, long NewLen);
    CWordLink();
    virtual ~CWordLink();
};
```

Figure 12.13. The CWordLink Class.

To avoid having to search the list of **CWordLink** objects each time a word is requested, we will reorganize the list into an array of **CWordElement** objects. The **CWordElement** class is given in Figure 12.14.

```
class CWordElement
{
public:
    CWordElement & operator=(const CWordLink &x);
    long Length;
    long Position;
    CWordElement();
    virtual ~CWordElement();
};
```

Figure 12.14. The CWordElement Class.

The word list functionality will be encapsulated into the **CWordList** class, which will do most of the work. This class will be used with an external string variable. An external routine must parse the string variable and identify the position and length of each word. Once a word is located, the *AddElement* function of **CWordList** must be used to add the word into the **CWordLink** chain of objects. Once all words have been found, the *MakeArray* function must be called to reorganize the list of words into an array of words. The *ArrayDone* variable is used to insure that the *MakeArray* function is called once and only once. When a new line of text is supplied to the component, the existing **CWordList** object will be deleted and a new one will be created. The *GetLength* and *GetPosition* functions will not return legitimate values until the *MakeArray* function has been called.

```

class CWordList
{
public:
    // The array
    CWordElement * Array;
    long Count;
    // The linked list
    CWordLink * Tail;
    CWordLink * Head;
    // status
    BOOL ArrayDone;
    // Creation
    void AddElement(long NewPos, long NewLen);
    void MakeArray(void);
    // Access
    long GetCount(void);
    long GetLength(long Ix);
    long GetPosition(long Ix);
    // construction and destruction
    CWordList();
    virtual ~CWordList();
};

```

Figure 12.15. The CWordList Class.

The property design table for the Word Extractor Cache is given in Figure 12.16. The *Line* property is used to supply the line of text, while the *Word* property is used to extract individual words. The *WordCount* property can be used to determine the number of words in the line.

Property Design Table		
Name	Type	Function
Line	String Default=Empty	Provides a line of text to be parsed into words.
Word	String Array Default=Zero Size	Returns the individual words in the line. Restrictions: Read Only
WordCount	Long Integer Maximum=None Minimum=0 Default=0.	Returns the count of the number of words in the line. Restrictions: Read Only

Figure 12.16. The Word Extractor Property Design Table.

When a new value is assigned to the *Line* property, the *MakeList* function is called to create a new **CWordList** object. For the purposes of this example, a word is considered to be a string of alphanumeric characters. All other characters are ignored. (What's wrong with this?) The *MakeList* function is given in Figure 12.17.

```

void CWordExtractorCtrl::MakeList()
{
    if (WL != NULL)
    {
        delete WL;
        WL = NULL;
    }
    if (m_line.IsEmpty())
    {
        FireErrorMessage("Empty String Given to Control");
        return;
    }
    WL = new CWordList;
    long WordPos = 0;
    long WordLen = 0;
    for (int i=0 ; i<m_line.GetLength() ; ) // work until EOL found
    {
        // Skip any "white space"
        while (i<m_line.GetLength() && !isalpha(m_line[i])
            && !isdigit(m_line[i]))
        {
            i++;
        }
        WordPos = i; Found beginning
        WordLen = 0;
        // Find the end of a the word
        while (i<m_line.GetLength() && (isalpha(m_line[i])
            || isdigit(m_line[i])))
        {
            WordLen++;
            i++;
        }
        if (WordLen > 0) // We may have found EOL
        {
            WL->AddElement(WordPos,WordLen);
        }
    }
    WL->MakeArray();
}

```

Figure 12.17. The MakeList Function.

The implementation of the *Word* property is given in Figure 12.18. This property acts like a simple array, however if there is an error in the request, the *ErrorMessage* event is fired, and the empty string is returned.

```
BSTR CWordExtractorCtrl::GetWord(long Idx)
{
    CString strResult;
    if (WL == NULL)
    {
        FireErrorMessage("No data to return");
        strResult.Empty();
    }
    else
    {
        long Pos = WL->GetPosition(Idx);
        long Len = WL->GetLength(Idx);
        if (Pos == -1 || Len == 0)
        {
            FireErrorMessage("Word Not Found");
            strResult.Empty();
        }
        else
        {
            strResult = m_line.Mid(Pos,Len);
        }
    }
    return strResult.AllocSysString();
}
```

Figure 12.18. The GetWord Function.

12.7. The Simple Graphical Editor Cache

The SGE Cache is used to hold several drawings created by the Simple Graphical Editor. Each drawing is identified by name, and can be retrieved explicitly by its name. It is possible to enumerate all items in the cache, and to selectively delete items from the cache. The items contained in the cache can be obtained from a number of different sources, including the SGE editor, the SGE background editor, and the SGE serializers. Internally the SGE Cache maintains a list of name-value pairs which consist of a name and a pointer to a **CGraphicList** object. The **CGraphHolder** object given in Figure

12.19 is used to maintain a name-value pair. The name-value pairs are stored in a simple linked-list. The list is doubly linked because name-value pairs can be selectively deleted.

```
class CGraphHolder
{
public:
    // linking: doubly linked because
    // we anticipate deletations from the middle of the chain
    CGraphHolder * Next;
    CGraphHolder * Prev;

    // The object and its name
    CString Name;
    CGraphicList * Item;

    // Construction and destruction
    CGraphHolder();
    virtual ~CGraphHolder();
};
```

Figure 12.19. The CGraphHolder Object.

As with our other examples, the linked list of name-value pairs is encapsulated in a header object that performs most of the work. This object, **CHolderList** is given in Figure 12.20. This object has three data items for maintaining the list, *Head*, *Tail*, and *Count*. The *Current* item is used during enumeration operations. The *AddItem* function is used to add new items to the list. It returns an error code if the item already exists. Items can be deleted from the list using either the name of the object, or the address of the **CGraphicList** object. There are two ways to delete an item from the list. The *DeleteItem* functions remove an item from the list and destroy the associated **CGraphicList** object. The *RemoveItem* functions will remove the item from the list without destroying the **CGraphicList** object. All of these functions will return an error code if the item cannot be found.

If one knows the name of an object, the *Find* function can be used to retrieve the address of the associated **CGraphicList** object. Similarly, if one has the address of a **CGraphicList** object, one can use the *GetName* function to retrieve the name of the object. These functions return NULL, or the empty string, if the item cannot be found.

All objects in the Cache can be enumerated using the *FindFirst* and *FindNext* functions. These functions use the *Current* item to enumerate the items in the Cache. *FindFirst* sets *Current* equal to *Head*, while *FindNext* sets *Current* equal to *Current->Next*. No other forms of enumeration are provided. Once *Current* has been set, the functions *GetCurrentName* and *GetCurrentAddress* can be used to retrieve the name and address of the current object. *FindFirst* and *FindNext* will return an error code if *Current* does not contain a valid object. (This usually indicates the end of the list.) If *Current* is not valid, *GetCurrentName* will return the empty string, and *GetCurrentAddress* will return NULL. The function *GetItemCount* can be used to obtain a count of the number of items currently being stored in the linked list.

The **CHolderList** object also has two utility functions, both of which are named *FindItem*.

```

// Main list-handler object
class CHolderList
{
protected:
    // List pointers and item count
    CGraphHolder * Head;
    CGraphHolder * Tail;
    long Count;

    // current iterator item
    CGraphHolder * Current;

public:
    // construction and destruction
    CHolderList();
    virtual ~CHolderList();

    // adding items
    long AddItem(LPCTSTR NewName, CGraphicList * Address);

    // removing items
    long RemoveItem(CGraphicList * RemAddress);
    long RemoveItem(LPCTSTR RemName);

    // removing items with model delete
    long DeleteItem(CGraphicList * DelAddress);
    long DeleteItem(LPCTSTR DelName);

    // searching by name and address
    CGraphicList * Find(LPCTSTR FindName);
    CString GetName(CGraphicList * FindAddress);

    // iterators and accessor functions
    long FindFirst(void);
    long FindNext(void);
    CString GetCurrentName(void);
    CGraphicList * GetCurrentAddress(void);
    long GetItemCount(void);

protected:
    // Finds in anticipation of further manipulation
    CGraphHolder * FindItem(CGraphicList * FindAddress);
    CGraphHolder * FindItem(LPCTSTR FindName);
};

```

Figure 12.20. The CHolderList Class.

Four error codes are used in the implementation of the **CHolderList** class. These are **SGE_NO_ERROR**, **SGE_DUPLICATE_NAME**, **SGE_NAME_NOT_FOUND**, and **SGE_ADDRESS_NOT_FOUND**. Most of the functions of the **CHolderList** class are straightforward, so we will provide only a few representative examples. Figure 12.21

gives the implementation of the *AddItem* function. This function searches the list for the new name, and if it is not found, creates a new name-value pair from its operands. The new pair is added to the tail of the linked list.

```
long CHolderList::AddItem(LPCTSTR NewName, CGraphicList *Address)
{
    // check for duplicate name
    if (FindItem(NewName) == NULL)
    {
        // if new name is unique
        // create new graph holder and fill data fields
        CGraphHolder * Temp = new CGraphHolder;
        Temp->Item = Address;
        Temp->Name = NewName;
        // link graph holder into list
        if (Head == NULL)
        {
            Head = Temp;
        }
        else
        {
            Tail->Next = Temp;
        }
        Temp->Prev = Tail;
        Tail = Temp;
        // All OK
        return SGE_NO_ERROR;
    }
    else
    {
        // Error
        return SGE_DUPLICATE_NAME;
    }
}
```

Figure 12.21. The AddItem Function.

Figure 12.22 shows the implementation of the *Find* function. A simple linear search is done to locate objects. If it is anticipated that a very large number of objects will be present in the cache, a more efficient method of organizing objects should be used.

```
CGraphicList * CHolderList::Find(LPCTSTR FindName)
{
    // Linear search for specified name
    for (CGraphHolder * Temp = Head ; Temp != NULL ; Temp = Temp->Next)
    {
        if (Temp->Name == FindName)
        {
            // terminate loop early if name is found
            return Temp->Item;
        }
    }
    // name not found
    return NULL;
}
```

Figure 12.22. The Find Function.

The *RemoveItem(LPCTSTR)* function is given in Figure 12.23. This function searches the list for the name, and if the item is found, it delinks the name-value pair and destroys it, without destroying the object pointed to by the name-value pair.

```

long CHolderList::RemoveItem(LPCTSTR RemName)
{
    // find the graph holder containing the specified name, if any
    CGraphHolder * Temp = FindItem(RemName);
    if (Temp == NULL)
    {
        // error
        return SGE_NAME_NOT_FOUND;
    }
    else
    {
        // unlink found item
        if (Temp->Prev == NULL)
        {
            Head = Temp->Next;
        }
        else
        {
            Temp->Prev->Next = Temp->Next;
        }
        if (Temp->Next == NULL)
        {
            Tail = Temp->Prev;
        }
        else
        {
            Temp->Next->Prev = Temp->Prev;
        }
        // delete the holder, but not the model in it
        delete Temp;
        return SGE_NO_ERROR;
    }
}

```

Figure 12.23. The RemoveItem(LPCTSTR) Function.

The *FindNext* function is given in Figure 12.24. This function first checks *Current* to see if it is non **NULL**, and if so then advances the pointer to the next item. If the result is a **NULL** pointer, the function returns zero, otherwise it returns one. The *FindFirst* function must be called first for the *FindNext* function to operate properly.

```

long CHolderList::FindNext()
{
    // Go to next item, if any
    // Return 0 if End of List found, return 1 if object is usable
    if (Current == NULL)
    {
        return 0;
    }
    Current = Current->Next;
    if (Current == NULL)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

```

Figure 12.24. The FindNext Function.

The properties and methods of the SGE Cache are essentially wrapper functions for the **CHolderList** object. Rather than providing formal definitions for the properties and methods of the component, we will simply list them and indicate the associated function of the **CHolderList** class.

Name	Type	Associated Function
AddItem	Method	<i>AddItem</i>
CurrentAddress	Read-Only Property	<i>GetCurrentName</i>
CurrentName	Read-Only Property	<i>GetCurrentAddress</i>
DeleteAddress	Method	<i>DeleteItem(CGraphicList *)</i>
DeleteName	Method	<i>DeleteItem(LPCTSTR)</i>
Find	Method	<i>Find</i>
FindFirst	Method	<i>FindFirst</i>
FindNext	Method	<i>FindNext</i>
GetName	Method	<i>GetName</i>
ItemCount	Read-Only Property	<i>GetItemCount</i>
RemoveAddress	Method	<i>RemoveItem(CGraphicList *)</i>
RemoveName	Method	<i>RemoveItem(LPCTSTR)</i>

Figure 12.25. Property/Method Association Table.

12.8. A Review of the Methodology.

The design of a Cache components has four elements, specifying the input view of the data, specifying the output view, designing the internal storage mechanisms, and specifying the access methods for the data. Specifying the input view of the data is essentially specifying the type of the data that will be placed in the component. Typically a single type of data will be input, strings, integers, floats, object of a specific type, and so forth. Specifying the output view is a little more complicated. Naturally, it is necessary to specify the type of the output data, but one must also define the relationship between the input data and the output data. Will the input items be passed through the Cache unchanged, or will they be transformed or combined in some way? Will they be disassembled into their individual parts, or combined with other data items to create more complex objects? These questions must be answered when specifying the output view of the data.

Next, it is necessary to specify the access methods of the component. Will data items be accessed sequentially or randomly? If they are accessed sequentially, in what order will they be accessed? If they are accessed randomly, how will one specify the desired item? Will a name of some sort be used, or will access be by an index number? If an index number is used, in what order will the items be stored? Once these questions have been answered, it is possible to proceed to the design of the internal storage mechanisms.

In one sense, the Cache is simply a wrapper for its internal storage mechanisms. The design of the input and output views as well as the design of the access methods simply gives information about how the internal storage mechanisms must be organized.

The interface structure of the Cache is quite simple, usually consisting of nothing more than a *Value* property. If the Cache supports several different access methods, there may be multiple *Value* properties, and perhaps some configuration properties and methods.

12.9. Conclusion

A Cache is particularly useful when it is necessary to manage a diverse collection of objects of the same type. The Cache provides a convenient place to store and organize data. The internal design of most Caches is relatively simple, so specialized Caches can easily be developed for specific applications.

12.10. Exercises

1. Create a cache that allows you to save E-Mail messages. An E-Mail message will have four parts, date and time of receipt, sender's name, subject, and content. You should be able to sort by date/time or by sender's name, and retrieve messages in either order.
2. Create a cache that will store a list of strings (great sayings, insults, jokes, etc.) and permit you to retrieve them in random order.
3. Create a cache that will allow you to store a list of lines. A line consists of two points, a point consists of two integer coordinates, which may be negative. The lines must be retrieved in descending order by length.