WORKING WITH DOCUMENTS AND VIEWS

Topics in this Chapter

Chapter 7

At the core of an MFC application is the concept of a *document object* and a corresponding *view window*. The document object usually represents a file the application has opened, while the view window provides a visual presentation of the document's data and accepts user interaction. The relationship between documents and views is a one-to-many relationship. In other words, a document can have many views, but you can associate a view with only one document.

Within your applications, you will represent document objects within classes that you derive from the MFC `CDocument` base class. You will derive your view window classes from the MFC `CView` class. In this chapter, you will learn about `CDocument` and `CView` and how to use them with simple, single-document interface (SDI) and multiple-document interface (MDI) applications.

7.1 Understanding the Two Document-Interface Structures

The built-in power of the document/view architecture, therefore, is that as users work with the application, they create and destroy instances of the file and user interface management code (and data) that define their very perception of the data with which they work.

A user-friendly application gives views to the application's data that make more sense to the user. For example, for most users it is probably difficult to imagine how cold it will be in South Dakota based on tabular temperature data from the entire United States. However, the common "blue is cold, orange is hot" weather graphs that appear in newspapers make it easy to guess what range of temperatures a traveler might expect with a quick glance at the right part of the map. The tabular data still has value, though. It's an easy way to enter the data in the first place, and it's the only way you might ever find out what the weather is like in South Dakota if you weren't completely sure where it was.

Core Note

Your applications may associate more than one instance of a particular view with a given document, and you may even associate instances of different views with the same document.

Single-document interface applications that you use AppWizard to produce only ever use one document and one view type, and only ever instantiate one of each of these classes. However, this is only true of the AppWizard-generated code — after the AppWizard creates your project, you can add as many views (and for that matter, documents) as you desire, if you decide that it's convenient to use multiple instances of each different view or document.

Multiple-document interface applications will make use of at least one document/view pair, but they may make use of additional documents and views in different combinations to enable the user to work with other files or to represent data in may different ways. As noted previously, you will learn more about multiple-document interface applications later in this chapter. Figure 7-1 shows which classes may support a simple SDI application that you implement using MFC objects.

In AppWizard-generated SDI applications, the `CMainFrame` class implements the frame window. In such cases, AppWizard will define the `CMainFrame` class for you in the Mainfrm.h header file and implement the class in the MainFrm.cpp source file. The `CMainFrame` class derives most of its functionality from the `CFrameWnd` class, which is the MFC wrapper class for a simple window. The `CFrameWnd` class does not do much in the single-document interface application. The notable exceptions are if you have added a status bar or dockable toolbars to the application, the `CMainFrame` class will handle the creation and initialization of those objects.

ch07.fm Page 195 Wednesday, November 24, 1999 11:41 AM

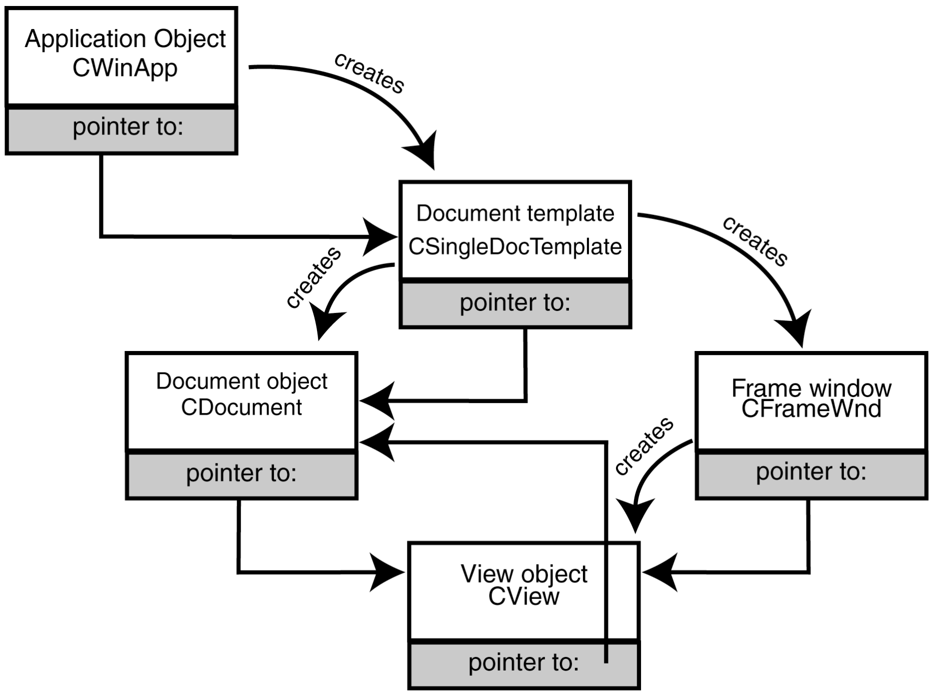


Figure 7-1 The model of the relationship between the five base classes for an SDI application.

While designing SDI applications is a useful exercise, SDI is generally only acceptable when you create small, simple applications. Within most applications that you design for actual use, whether for public use or within your own organization, you will require the ability to use multiple views, and generally multiple documents, to organize information. When you use multiple documents within a single application, you will use the multiple-document interface (MDI) model for the document/view structure. The layout of a multiple-document interface application is a little more complicated than the layout of a single-document interface application. Figure 7-2 shows the basic layout of the MDI application structure.

As you can see from Figure 7-2, multiple-document applications still use a main frame that holds the menu, toolbar, and status bars. However, in the MDI application, the `CMainFrame` class derives from MFC's `CMDIFrameWnd` class, instead of the `CFrameWnd` class. `CMDIFrameWnd` has the same visual characteristics as `CFrameWnd`, but it also implements the MDI frame protocol that Windows expects in an MDI application.

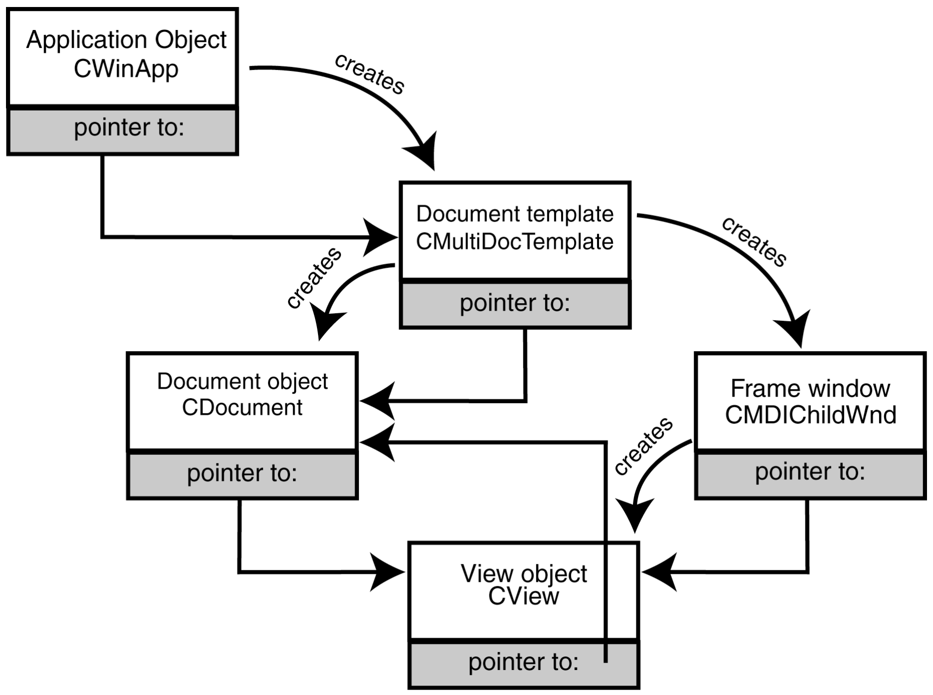


Figure 7-2 The basic layout of the MDI-application object structure.

The child windows that the figure depicts are also frame windows, but the child windows are instances of the MFC class `CMDIChildWnd`. This MFC class provides the child window that Windows MDI applications use in their client area to hold each instance of the MDI application's views. The frameworks will create one `CMDIChildWnd` to contain each view the application needs, just like the `CMainFrame` object wrapped the single view in the SDI application. The wrapped view may be of any type and can refer to any open document that the application is currently managing.

As a developer of MFC applications, it is your responsibility to decide exactly what kind of documents and views you will implement, and how they'll interact with the basic framework provided by MFC's implementation of the single-document or multiple-document interfaces. Your code alters and enhances the way the generic documents and views interact and behave. By tuning things to work the way you want them to, you will eventually develop the skeleton MFC provides into an application that does exactly what you need. Over the course of this chapter, you will learn more about the components of the MDI application and use a simple MDI application so that you can better decide what structure your applications require.

Conventional Windows applications written in C (and using the Windows API) would modify the way that Windows' own classes work. Within the application's body, your program code then would paint, draw, or store something as a direction to input messages (or combinations of input messages) to your application's windows. Fortunately, by using MFC, you are able to focus more closely on working with classes, rather than working with a series of system function calls. These MFC classes also support the ability to intercept those basic Windows messages and, when appropriate, do work at a much lower level.

7.2 Complex Combinations of Documents, Views, and Frame Windows

As you have learned, the standard relationship among a document, its view(s), and its frame window(s) is described in a relatively straightforward manner: a one-to-many relationship between documents and views, and a one-to-one relationship between each view and a frame window. Many applications need only to support a single document type (but possibly let the user open multiple documents of that type) with a single view on the document and only one frame window per document. But some applications may need to alter one or more of those defaults — creating multiple views on a single-document type, single views on multiple-document types, or multiple views on multiple-document types. It is worthwhile to consider the different situations that you may encounter when working with documents and views and how you should design your applications to respond appropriately.

7.3 Working with Multiple-Document Types

Whether you create an SDI or an MDI application, AppWizard will create only a single document class for you. In some cases, though, you may need to support more than one document type. For example, your application may need both worksheet and chart documents. Your application will probably represent each document type with its own document class and typically by its own view class or classes as well. When the user chooses the File menu's New option, the framework will display a dialog box that lists the application's supported document types. After the user chooses a document type, the application creates a document of that type. The application then manages each document type with its own document-template object.

To create extra document classes within your own applications, use the Add Class button in the ClassWizard dialog box. Choose `CDocument` (or `COLEDocument`) as the Class Type to derive form and supply the requested document information. Then implement the new class' data structures.

To let the framework know about your extra document class, you must then add a second call to `AddDocTemplate()` in your application class' `InitInstance()` member function. For example, an application with two documents would include code within its `InitInstance()` member function similar to the following:

```
CMultiDocTemplate* pDocTemplate;
PDocTemplate = new CMultiDocTemplate(
    IDR_OPAINTTYPE,
    RUNTIME_CLASS(CSample1Doc),
    RUNTIME_CLASS(CMDIChildWnd),
    RUNTIME_CLASS(CSample1View));
AddDocTemplate(pDocTemplate);
pDocTemplate = new CMultiDocTemplate(
    IDR_OPAINTTYPE,
    RUNTIME_CLASS(CSample2Doc),
    RUNTIME_CLASS(CMDIChildWnd),
    RUNTIME_CLASS(CSample2View));
AddDocTemplate(pDocTemplate);
```

7.3.1 Understanding the CDocument Class

The MFC-provided `CDocument` class provides the basic functionality for your application's document objects. The `CDocument` class' basic functionality includes the ability to create new documents, serialize document data, provide basic cooperation between a document object and view window, and more. MFC also provides a series of `CDocument`-derived classes that implement functionality specific to certain application types. For example, MFC provides the `CRecordset` and `CDAORecordset` types to simplify the creation of database views. You can visualize the relationship between documents and views as shown in Figure 7-3.

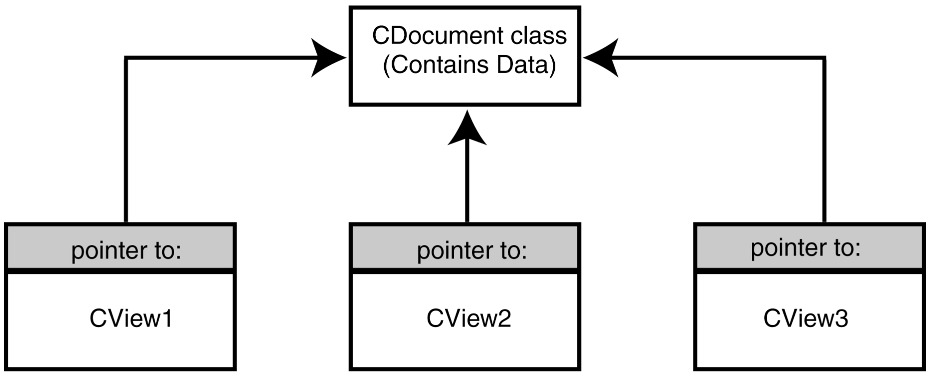


Figure 7-3 The one-to-many relationship between a document and its views.

7.3.2 Declaring a Document Class in Your Application

When you use the AppWizard to create your applications, you often do not need to worry about declaring your base document class — the AppWizard does it for you. However, it is still useful for you to understand the behavior of the `CDocument` class because more complex programs might very possibly require multiple instances of multiple derivations from `CDocument`. In addition, understanding the behavior of `CDocument` lets you easily enhance the AppWizard-generated application skeleton.

Core Note

Whether you create a single-document interface application or a multiple-document interface application, the AppWizard will only derive a single-document class for you from the MFC `CDocument` base class.

When you build a simple MFC application, it is often enough for you to make relatively minor modifications to your AppWizard-supplied document class. Often, you will not need to do much more to the class than add some member variables and some member functions that other portions of the program can use to access those member variables.

For example, the document object for a simple communications program (such as a terminal emulator) might contain member variables for settings. Those member variables would probably store information such as a telephone number, speed, parity, number of bits in each transmission segment, and so on. You could easily represent the communications settings with a set of simple member variables in the derived document class, as shown in the following code snippet:

```
class CSimpleTermDoc : public CDocument {
  protected:
    CSimpleTermDoc();
    DECLARE_DYNCREATE(CSimpleTermDoc)
  public:  CString_m_sPhoneNum;
    DWORD m_dwTransSpeed;
    WORD m_nTransParity;
    WORD m_nTransBits;
    DWORD m_dwConnectTime;
```

After you declare the member variables, you must ensure that the program initializes the variables to some default values in the `CSimpleTermDoc` class' `OnNewDocument()` member function. In addition, you must place code in the `Serialize()` function to ensure that the program serializes the variables properly.

For your simpler applications, you really need do nothing beyond the initialization and serialization of your member variables to have a complete, fully functioning document class.

7.3.3 Using CDocument's Member Functions

The `CDocument` class has several member functions, in addition to the serialization and initialization member functions that your applications will frequently use. The first set of member functions provides access to the associated view subjects. Every document object that you use within your applications will have a list of view objects that you associate with it. You can call the `GetFirstViewPosition()` member function for the document object to obtain an iterator to this list. The iterator will be of type `POSITION`.

You will use values of type `POSITION` throughout the MFC, primarily with collection classes. When your applications must traverse a list, you will typically request an iterator that the collection class associates with the first object on the list, and then use an iterator function to access the actual elements the list contains, one-by-one. `CDocument`, in this context, is a collection class; it maintains information about the collection of views associated with the class. Therefore, after you obtain the iterator to the first view of the `GetFirstViewPosition()` member function, you can repeatedly call the `GetNextView()` member function to work through the remaining views in the collection.

In other words, to process all the views that your program has associated with a given document object, your program code will generally look similar to the following:

```
POSITION posView = GetFirstViewPosition();
while (posView != NULL) {
  CView *pView = GetNextView(posView);
  // Do something with the pointer to the view
}
```

However, if all your program code is trying to accomplish is to notify all the views for the document that information within the document has changed, you can simply invoke the document object's `UpdateAllViews()` member function instead of iterating the views. Furthermore, you can also specify application-specific data that instructs the views to selectively update only portions of the view windows when you call the `UpdateAllViews()` function.

Some other view-related member functions for the document object that you will use much less frequently include the `AddView()` and `RemoveView()` functions. As their names indicate, the functions let you manually add and remove views from a document's list of views. In general, you will use the functions only rarely, as most developers simply use the default MFC implementation with little or no modification.

Whenever the document's data changes (either through a user's action or through internal program processing), your program should call the `SetModifiedFlag()` member function. Consistent use of `SetModifiedFlag()` will ensure that the MFC framework prompts the user before letting the user destroy an unsaved, changed document. Should you decide to override the framework, you can call the `IsModified()` member function to obtain the status of the flag.

You can use the `SetTitle()` member function to set the document object's title. The application, in turn, will display the title you set in the frame window (the main frame window in an SDI application, and the child frame window for the object in an MDI application).

You can also set the fully qualified path name for the document with the `SetPathName()` function and obtain the path name with the `GetPathName()` function. Finally, you can obtain the document template that the program associated with the document at the document's creation through a call to `GetDocTemplate()`.

7.3.4 Better Understanding Documents and Message Processing

One of the most important features of a document is that a `CDocument` object is not directly associated with a window. Instead, a `CDocument` object is itself a command-target object — which means that the object can receive messages from the operating system. The view objects that you associated with a `CDocument` object are responsible for routing messages from the operating system to the document.

Because the view objects you associate with the document and the frame window that holds the document will receive messages before passing them through to the document, you have a great deal of control over which messages the frame window, views, and document process. However, there are some common-sense rules of thumb (as well as some simplicity issues) that provide you with a good starting point for how to process incoming messages.

When you consider messages, or for that matter any time that you are working with the document-view architecture, you should always keep in mind that a document is an abstract representation of your data — a representation, that is, which is independent of the visual representation of the data that the view window will provide. As importantly, a document may have one, many, or no views attached to it, so documents should respond only to messages that are global in nature. That is, a document should respond only to messages that have an immediate effect on the document's data, which messages' effect all the views attached to the document should reflect. On the other hand, views should respond to messages that are specific to that window's view only.

In practical terms, the division of responsibilities between documents and views generally makes it easier to determine how to process a given command. For example, if your application has a Save command, which the user would select to save the data in the object, the document should handle that command because the command is concerned with the data, not how the user sees the data.

On the other hand, if your application supports a Copy command, which the user would usually select to copy data the user has selected within the display, you would probably want to handle the command in the view. In fact, if a document supports multiple views, the data selected in each view might vary — making it even more clear that you should generally process the copy command separately for each view attached to a document.

Both cases we have considered already are relatively clean-cut — you are saving the data in the first example, and you are copying representation of the data from within one view to another view in the second example. However, there are some borderline cases. A common one is the Paste command. Determining whether the document class or the view class should handle it is slightly more complex. The Paste command impacts the entire document (you are inserting data into the document), not just a single view. On the other hand, the current view may have significant importance when pasting information into a document. For example, the paste action may actually replace existing, selected text within the view. In other words, the decision you must make about whether the document object or the view object should handle actions of this type is dependent on your application's design, and is usually something you should think through carefully.

Just to keep it interesting, there are also certain commands that you should not handle in either the document class or the view class, but rather in the frame window's code. Excellent examples of commands that you should handle within the frame window include commands to hide and display toolbars. The presence or absence of the toolbar is not particularly material to a document or its views. Rather, it is a configuration issue with effects global to the entire application.

7.3.5 Overriding Virtual Document Functions

As you learned earlier in this chapter (when working with `OnNewDocument()` and `Serialize()`), many of the member functions the `CDocument` class defines are virtual functions, meaning that you can override them in your own class declarations. The virtual functions in the `CDocument` class provide default processing that is sufficient for most needs. However, you will also find that your programs must perform specific processing for a certain document that the default processing does not provide.

For example, the `CDocument` class and its derivatives will call the `OnNewDocument()` member function whenever the program initializes a new document object (or when the program reuses an existing document object in an SDIU application). Your applications will typically call the `OnNewDocument()` function when handling a File New command. Similarly, your `CDocument` calls the `OnCloseDocument()` member function when the application is about to close a document. You should override this document within your own document classes if your application must perform any clean-up operations before destroying the document object.

Your document classes will call the `OnOpenDocument()` and `OnSave-Document()` functions to read a document from disk or to write a document to disk, respectively. You should override these functions only if the default implementation (which calls the `Serialize()` member function) is not sufficient. An excellent example of a situation in which you would override `OnOpenDocument()` and `OnSaveDocument()` is if you are encrypting data before you write it to the disk and decrypting it when you reload it from the disk.

The default implementations of both `OnOpenDocument()` and `OnCloseDocument()` call the `DeleteContents()` member function. The `DeleteContents()` member function deletes the document's contents without actually destroying the document object. Using `DeleteContents()` when opening a new document is more efficient (in terms of both memory usage and application speed) than actually closing and destroying the original document object and creating a new document object.

The `OnFileSendMail()` member function sends the document object as an attachment to a mail message. It first calls `OnSaveDocument()` to save a copy of the document to temporary disk file (in the directory set by your TEMP environment variable). Next, the program code within the member function attaches the temporary file to a MAPI message. The member function uses the `OnUpdateFileSendMail()` member function to enable the command that you identify with the constant `ID_FILE_SEND_MAIL` in the application's menu or remove it altogether if MAPI support is not available to the program. Both `OnFileSendMail()` and `OnUpdateFileSendMail()` are overridable functions, which lets you (relatively easily) implement customized messaging behavior within your applications.

7.4 Working with Complex Document Data

Earlier in this chapter you learned how to derive simple document classes from `CDocument`, within which you can store the document's data in a series of simple member variables. However, creating applications that you will use in the real world tends to be more demanding. Most applications you will develop will require significantly more advanced data than what you could ever possibly represent with a few variables of simple data types.

There are many different approaches that you will use to manage complex data types within a document object; however, arguably the best approach is to use a set of classes that you derive from the `CObject` class. Each derived class, then, will store the complex data objects. The document, in turn, will use a standard or custom-created collection class to embed the objects within the document class. For example, you might create data definitions similar to the following for an application:

```
class CAppObject : public CObject {
  // definitions
}

class CAppSubObject1 : public CObject {
  // definitions
}

class CAppSubObject2 : public Cobject {
  // definitions
}
```

Then, within the declaration of the document class, you would include a `CObList` member. The `CObList` class supports ordered lists of nonunique `CObject` pointers accessible sequentially or by pointer value. `CObList` lists behave like double-linked lists. Your document declaration, therefore, would look similar to the following:

```
class CSampleDoc : public CDocument {
  // code here
  public
    CObList_m_DataObList;

  // code here
}
```

In a complex situation such as the one just outlined, it is often not sufficient to simply declare member variables. Your document class is also likely to require member functions that provide methods to let views and other objects that must access the document's data do so. For example, you may not want to let other classes (such as a view class) directly manipulate the `m_DataObList` variable directly. Instead, you should usually provide a member function that the view class can access to iterate through the `m_DataObList` object as it needs to.

Such member functions should also ensure that each time the document's data changes, the application properly updates all the document's views. The member functions should also call the document's `SetModifed` member function to indicate to the document that an accessing function or class has changed the document's data. If your application will support an undo-type capability, you should also place your application's buffered undo data into its correct storage location while inside the member function. To understand this better, consider the following member function, AddNewObj(), which adds a new object to the document's object list:

```
BOOL CSampleDoc : AddNewObj(CAppObject *pObject)
{
  try {
    m_DataObList.AddTail((CObject *)pObject);
    SetModifiedFlag(TRUE);
    UpdateAllViews(NULL, UPDATE_OBJECT, pObject;
    return TRUE;
  }
  catch(CMemoryException *e) {
    TRACE("Doc—AddNewObj_memory allocation error.\n");
    e->Delete();
    return FALSE;
  }
}
```

Understanding the importance of the AddNewObj() member function is easier when you consider the relationship between the document and its views and how the program will pass control back and forth between the two.

First, the user will interact with the view, which might result in a new object being added, an existing object being modified or deleted, or some other action. For now, presume that the user's actions result in the need to add a new object to the document. To add a new object, the view object calls the AddNewObj() member function. After the member function adds the new object successfully, the document object will call the UpdateAll-Views() member function, which, in turn, will call the OnUpdate() member function of each view that you have previously associated with the document. The AddNewObj() member function passes a hint to the UpdateAllViews() member function through the use of the application-defined UPDATE_OBJECT constant and a pointer to a CObject. The hint assists all the views in most efficiently updating their component windows by instructing the views to repaint only those regions of the view directly and indirectly affected by the addition of the new object. Figure 7-4 shows the control-passing mechanism that the views and the document will use.

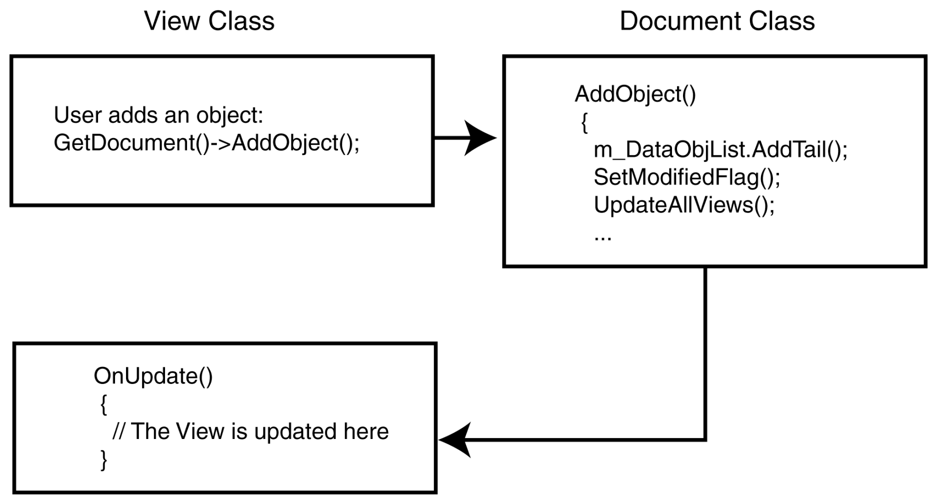


Figure 7-4 Control passes from the view class to the document class and back.

Another advantage of using MFC collection classes within your application is that collection classes support serialization. For example, to load and save your document's data that is stored in `CObject` objects and referenced through a `CObList` object, all you need to do is to construct the document's `Serialize()` member function as shown here:

```
void CSampleDoc::Serialize(CArchive &ar)
{
  if (ar.IsStoring()) {
    // serialize any non-collection class data here
  }
  else {
    // serialize any non-collection class data here
  }
  m_DataObList.Serialize(ar);
}
```

You should be aware, however, that for this technique to work you must implement the `Serialize()` member function for all your object classes. A `CObject`-derived class will not serialize itself. If you decide to use one of the general-purpose collection templates, serialization is an issue that you must pay close attention to. The collection `CArray`, `CList`, and `CMap` rely on the `SerializeElements()` member function to serialize the objects within the collection. MFC declares this function as shown here:

```
template <class TYPE> void AFXAPI
    SerializeElements(Carchive &ar,
    TYPE *pElements, int_nCount);
```

Because the collection class templates do not require that you derive `TYPE` from `CObject`, they do not call the `Serialize()` member function for each element (because the `Serialize()` member function is not guaranteed to exist). Instead, the default implementation of `SerializeElements()` performs a *bitwise* read or write action. However, as you can imagine, in most cases, a bitwise read or write is not what you will want to perform. Rather, you should implement your own `SerializeElements()` function for your objects. You might implement such a function as shown here:

```
void SerializeElements(CArchive &ar,
    CAppObject **pObs, int_nCoutn);
{
  for (int i = 0; i< nCount; i++; pObjs++)
    (*pObs->Serialize(ar);
}
```

7.4.1 Understanding the Benefits of CCmdTarget and CDocItem

As you learned in the previous section of this chapter, you can use objects that you derive from the `CObject` class to store data within your documents. Unfortunately, if you wish your documents and applications to support OLE automation, the `CObject` class is insufficient. Instead, you must declare your objects as *command targets.* If you wish to support OLE automation, you may prefer to derive your data from the MFC `CCmdTarget` base class.

Alternately, and usually better, you may want to derive your data objects from the MFC `CDocItem` class. You can either create a collection of `CDoc-Item` objects yourself or rely on MFC's `COleDocument` class to create the collection. In other words, rather than deriving your document class from `CDocument`, derive it from `COleDocument`. You can use `COleDocument` in OLE applications where either the `COleDocument` class or a class previously derived from `COleDocument` is the base class for the OLE application's document class. Like `CDocument`, `COleDocument` is a collection class. `COleDocument` supports a collection of `CDocItem` objects, which are in turn either `COleServerItem`- or `COleClientItem`-derived. However, `COleDocument` supports `CDocItem` generically (that is, it doesn't care whether the item is a server or client item). `COleDocument`'s generic implementation means that you can add your own `CDocItem`-

derived objects to the collection without fear that doing so will interfere with normal OLE operations and behavior.

One nice thing about working with `COleDocument` is that it adds additional `CDOcItem` members for you automatically. If you use `AddItem()`, `RemoveItem()`, `GetStartPosition()`, and `GetNextItem()`, you can add, remove, and retrieve document items without further processing. The underlying MFC coding handles your other needs (such as serialization) without further programming on your part.

However, working with `COleDocument` is not without its pitfalls. Because of how you derive your document items and the OLE `COleClientItem` and `COleServerItem` objects, you may need to perform certain special programming actions to add certain functions to a given object. For example, suppose that you declare your object items as shown here:

```
class CSampleDocItem : public CdocItem {
  // more code here
  CRect m_Rect;
}
```

In addition, suppose that you also support the `m_Rect` member variable within your OLE client items, as shown here:

```
class CSampleClientItem : public ColeClientItem {
  // more code here
  CRect m_Rect;
};
```

Given these two declarations, you might suppose that you can create a function that takes an item from your document and manipulates its `m_Rect` member as shown here:

```
void sampFunc(CDocItem *pItem) {
  samp2Func(pItem->m_Rect);   //Error!
}
```

Because the `CDocItem` class by itself does not contain an `m_Rect` member variable, the compiler will halt the program's compilation with an error at the function declaration. Unfortunately, using a pointer to your own `CDocItem`-derived class doesn't really solve the problem either:

```
void sampFunc(CSampleDocItem *pItem) {
  samp2Func(pItem->m_Rect);
}
```

While declaring the function in this manner will support your derived class, it won't support OLE client items of type `CDocItem` — a significant issue. An obvious solution is to simply create two overridden versions of sampFunc, but maintaining two separate, identical versions of the same function is not only inelegant, it makes maintenance all that much more difficult. The best solution is to instead create a wrapper function that takes a pointer to a CDocItem object and uses MFC runtime type information to obtain the member variable, as shown here:

```
CRect_GetRect(CDocItem *pDocItem)
{
  if (pDocItem->IsKindOf(RUNTIME_CLASS(CSampleDocItem)))
    return_((CsampleDocItem *)pDocItem)->m_Rect;
  else if (pDocItem->
      IsKindOf(RUNTIME_CLASS(CSampleClientItem)))
    ASSERT(FALSE);
  return CRect(0, 0, 0, 0);
}

sampFunc(CDocItem *pItem)
{
  samp2Func(GetRect(pItem));
}
```

This solution, however, does require that you declare and implement both the CSampleDocItem and the CSampleClientItem classes with the DECLARE_DYNAMIC and IMPLEMENT_DYNAMIC macros. In general, however, that should not be an issue, because your document objects will typically support serialization. When you declare and implement a class with the DECLARE_SERIAL and IMPLEMENT_SERIAL macros, the DECLARE_DYNAMIC and IMPLEMENT_DYNAMIC macros are implied.

7.5 Understanding How Your Applications Manage Documents and Views

Now that you have learned about documents, and been introduced briefly to document templates, it is important to understand how MFC keeps track of the documents and views that comprise the application, and which documents and views are related to one another.

As the document/view architecture is the cornerstone of any document-based application (as you learned earlier, dialog-based applications perform differently than document/view applications), MFC must be able to create and destroy objects from the document/view implementation classes. As your application may handle more than one type of document/view relationship, MFC must have some way of knowing which document, view, and display classes you implement, what the relationships are between the classes, and how to create the implementations of the classes at runtime. After all, while one document might support many different types of views, associating other views with that same document might be nonsensical.

7.5.1 Working with the CSingleDocTemplate Class

To learn how MFC describes and maintains these associations, use the App-Wizard to create a simple single-document application. When you do, you will find source code that looks like the following:

```
CSingleDocTemplate* pdocTemplate;
pDocTemplate = new_CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CSIMPLEDoc),
    RUNTIME_CLASS(CMainFrame),   // main SDI frame window
    RUNTIME_CLASS(CSIMPLEView));
AddDocTemplate(pDocTemplate);
```

The code dynamically allocates a new `CSingleDocTemplate` object. The constructor for `CSingleDocTemplate` takes four parameters. The first parameter is a resource ID. You will learn the significance of the resource ID later in this chapter. The second, third, and fourth parameters associated with the `CSingleDocTemplate()` constructor are pointers to runtime class information. The `RUNTIME_CLASS()` macro generates a pointer to the runtime class information for the application's document, main frame, and view classes. These pointers are all passed to the `CSingle-DocTemplate` constructor, which keeps the pointers so that it can create instances of the objects as needed to put together a complete document/view team.

The `CSingleDocTemplate` object lives as long as the application continues to execute. MFC uses the object internally and destroys any added document templates as the application's `CWinApp` object is destroyed. You can find the code that allocates `CSingleDocTemplates` to your application in its `CWinApp::InitInstance()` function, but you will never write code that deletes the document template objects if you use `AddDocTem-plate()` to add the template, because the `CWinApp` destructor function will clean up that document template allocation for you automatically.

7.6 Understanding the CMultiDocTemplate Class

Much as the `CSingleDocTemplate` class defines a document template that implements the single-document interface, the `CMultiDocTemplate` class defines a document template that implements the multiple-document interface (MDI). An MDI application uses the main frame window as a workspace in which the user can open zero or more document frame windows, each of which displays a document.

An MDI application can support more than one type of document, and documents of different types can be open at the same time. Your application has one document template for each document type that it supports. For example, if your MDI application supports both spreadsheets and text documents, the application will have two `CMultiDocTemplate` objects.

The application uses the document templates when the user creates a new document. If the application supports more than one type of document, then the framework gets the names of the supported document types from the document templates and displays them in a list in the File New dialog box. Once the user has selected a document type, the application creates a document class object, a frame window object, and a view object and attaches them to each other.

Core Note

You do not need to call any member functions of `CMultiDocTemplate` except the constructor. The MFC framework handles `CMultiDocTemplate` objects internally.

7.7 Working with Frame Windows

Throughout this chapter you have learned about documents and frames. However, as you have probably gathered already, understanding the importance of frames in a document/view application is also important, even though you will not work with the frame class(es) anywhere near as frequently as you will work with the document and view classes.

In fact, the view your application implements is a window, but not a pop-up or frame window. Instead, it is a borderless child window that doesn't have a menu of its own, so it must be contained by some sort of frame window. MFC places the view window you create into the client area of the frame window identified in the document template constructor. In an SDI application, the frame window is always the main window for the application. Similarly, the frame window for views within a multiple-document interface application is an MDI child window.

When developing a Windows application, most programmers will not take the extra step of separating the client area of their application from the frame window. Instead, you would typically create a `WS_OVERLAPPED`-style window and paint right in its client area. To make MFC a little more modular, Microsoft implemented it so that it makes a distinction between the two types of frame windows that you might use. That is, MFC makes both an internal and an external distinction between a single-document interface frame window and a multiple-document interface frame window. You will learn more about frame windows later, but for now, it's enough to understand that the frame window is the one that receives all of the menu and window frame messages.

7.7.1 Understanding the CMDIFrameWnd and CMDIChildWnd Classes

The `CMDIFrameWnd` class provides the functionality of a Windows multiple-document interface (MDI) frame window, along with member functions that you will use within your applications to manage the window. To create a useful MDI frame window for your multiple-document interface application, you must derive a class for the main frame window from `CMDIFrameWnd`. After you derive the class, you will add member variables to the derived class to store data specific to your application (but not data specific to an individual document). In addition, you must implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the windows, either by the operating system or by the document template.

You can construct an MDI frame window in two ways: by calling either the `Create()` member function or the `LoadFrame()` member function of `CFrameWnd()`. However, before you call `Create()` or `LoadFrame()`, you must use the C++ `new` operator to construct the frame window object on the heap. Before calling the `Create()` member function, you can also use the `AfxRegisterWndClass()` global function to register the window class and set the icon and class styles for the frame. You should use the

`Create()` member function to pass the frame's creation parameters as immediate arguments.

On the other hand, the `LoadFrame()` member function requires fewer arguments than the `Create()` member function, and instead retrieves most of its default values from resources that you create within the project, including the frame's caption, icon, accelerator table, and menu. To be accessed by `LoadFrame()` (and loaded into the definition for the new window), all these resources must have the same resource ID (for example, the MFC default resource ID `IDR_MAINFRAME` or any other resource ID such as `IDR_PARANTFRAME`).

Core Note

Although MFC derives the `CMDIFrameWnd` class from the `CFrameWnd` class, you do not need to use the `DECLARE_DYNCREATE` macro when you declare a frame window class that you derive from `CMDIFrameWnd`.

Similarly, the `CMDIChildWnd` class provides the functionality of a Windows multiple-document interface (MDI) child window, along with members for managing the window. An MDI child window looks much like a typical frame window, except that the MDI child window appears inside an MDI frame window rather than on the desktop. An MDI child window does not have a menu bar of its own, but instead shares the menu of the MDI frame window. The framework automatically changes the MDI frame window's menu bar to represent the currently active MDI child window's menu bar.

To create a useful MDI child window for your application, you must derive a class from `CMDIChildWnd` (or, if you do not intend to customize the window's actions, you can simply use the default `CMDIChildWnd` class). You will then add member variables to the derived class to store data specific to the document that the child window will be associated with within the application. Furthermore, you must implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window (otherwise, MFC will use the `CMDIChildWnd` class' default handlers to respond to messages the window receives). There are three ways to construct an MDI child window:

- Directly construct it using the `Create()` member function.
- Directly construct it using the `LoadFrame()` member function.
- Indirectly construct it through a document template.

Core Note

Unlike a frame window class that you derive from the `CMDIFrameWnd` class, a frame window class that you derive from `CMDIChildWnd` must be declared with the `DECLARE_DYNCREATE` macro for the `RUNTIME_CLASS` creation mechanism to work correctly.

The only time your applications will not create a view within a real frame window is when the view is active as an embedded OLE object. The view will still have a frame in such situations, but the frame will be very different from the standard frame windows you have learned about.

As you have learned, an SDI application usually creates a `CFrameWnd` instance, while an MDI application typically creates a `CMDIFrameWnd` instance as well as one or more `CMDIChildWnd` instances. On the other hand, if you've written a dialog-based application, the dialog is the main window and your application doesn't have any frame window.

Due to the way command dispatching works, you will find that the frame window often acts as a catch-all for choices in your command window. In other words, any command from a menu that isn't handled by your view will be offered to your frame window for it to handle.

You should implement handlers, ready for any frame message, no matter which view the user is currently working with. If you have menu choices that should react in different ways for different views, you can implement handlers in both the frame and the view classes. The view handler will be executed if the view object is active; otherwise, the frame's handler will be called.

7.7.2 Understanding the Role of AfxGetMainWnd()

Frames act as the main window for the thread that controls your process. If you call `AfxGetMainWnd()` at any point in your program, you can retrieve a pointer to the `CWnd` class that your application uses as its main window. You will need to cast that pointer to the appropriate type if you need to access any `CFrameWnd` or `CMDIFrameWnd` specific members.

A frame window is responsible for one or two more things than just making sure your application has a menu and a sizeable frame. It also serves as an anchor for your window's toolbar and status bar. If your application has a status bar or a toolbar, you will find code in your main frame window that creates instances of `CStatusBar` or `CToolBar`. As you might guess, `CStatusBar` creates a status bar and `CToolBar` handles a toolbar. In most applications, the creation of these windows is handled in the `OnCreate()` member of the application's frame window.

Core Warning

Since the C++ objects are members of the frame windows, your application will create them at the same time that it creates the frame window object.

If you are working with an AppWizard-produced application, the status bar in your application will be called `m_wndStatusBar` and your first toolbar will be called `m_wndToolBar`. Note that it is your *first* toolbar that receives this name. Your frame is completely capable of handling more than one toolbar — in fact, MFC will layout as many toolbars as you would like.

By the way, `CStatusBar` and `CToolBar` classes are dependent on a frame window. Using them in other types of windows (such as dialogs) is beyond the scope of this book. Doing so is not usually something that most programmers would do because it results in a nonstandard interface. These two classes are not really dependent on the document/view architecture, but they do rely on the frame window associated with the document and view classes. The classes use the frame window to lay themselves out in your application's user interface and they keep the view informed of the area it has available to draw in.

7.8 Understanding the Document Template Resources

As you learned previously in Section 7.5.1, the first parameter to the `CSingleDocTemplate` constructor is a resource ID. The first parameter tells the frame that the template will use what kind of resources it must have available to complete the link. This ID identifies the resources used to supply the frame with an accelerator table, menu, and icon. The frame window that your application uses should have the same resources ID for each resource type it wishes to use.

If you examine `HexView.rc`, you will find that there's an accelerator table, a menu, and an icon, each with the ID of `IDR_MAINFRAME` that corresponds to the ID the application passes to the `CSingleDocTemplate` constructor. Having exactly the same frame window resource IDs is far more convenient than requiring the constructor for `CSingleDocTemplate` to take six or seven parameters.

The resource ID is also the ID of an entry in your application's string table. The identified string has a very special format: it's really seven strings in one, each separated by a newline (`\n`) character.

7.8.1 Considering the Document Template Lifecycle

As you might imagine, `CSingleDocTemplate` is a lightweight class — that is, it takes very little memory. You shouldn't worry about keeping document template classes lying about, even if you have dozens of them.

`CSingleDocTemplate` and `CMultiDocTemplate` are used heavily by the application frameworks. After setting them up and getting them started, your application will depend on the templates to manage the document, view, and frame windows objects — but you will no longer need to manipulate the templates directly. As you saw earlier in this chapter, your application should register all of the document templates that it will use during its `CWinApp::InitInstance()` member function. By making them public members of your application object, you can access them later on when you need to juggle documents and views.

When you think about and manipulate document templates, probably the best single perspective for you to maintain is to let MFC do the work. In other words, you should ask MFC's code in the document template object to create the view and document you need, and hook up all of the associations. In a well-designed application, you should almost never have to directly create your own views, documents, and frames. The best applications let the document template do the work.

7.8.2 Advanced Work with Templates

Now that you understand the basics of templates, it is important to think about them from the perspective of application design. The way your application works really depends on your point of view. If your users have to do a lot of work to get to a view of the data they are interested in, they'll quickly get frustrated. Worse yet, if the views your application offers do not represent information in the way your users perceive as intuitive, your application will be seen as awkward, since the users will have to spend too much time thinking about how things should work instead of actually getting work done.

Out-of-the-box application frameworks that the AppWizard produces are too good not to use for most of your applications. Even after some modification, MFC will react to your changes in ways that are generally seen by the user as intuitive and consistent with the interfaces they are used to.

If you register several different document templates, you will get the extra dialog box to allow the user to choose their document type after selecting the File menu's New option. And once you have taken advantage of this simple opportunity, you will soon find that there are several other instances where you might want your application to differ slightly from the mainstream.

7.8.3 Working with Multiple Templates

You are always allowed to use more than one template when you are running an MFC document/view application. In general, your application will create any necessary templates as it handles `CWinApp::InitInstance()`. If your application initially came from the AppWizard, you will find that the function has been coded to create and register a template for the document/view pair that your application uses by default.

If you ever need to use documents or views in any other combination, you should add code to create a template for those particular document/view pairs. Doing so will make it much easier for you to create instances of the pairs at the user's request. The `CDocTemplate`-derived object you create is just that — an object — and as such, you will need to maintain a pointer to it after you use `new` to create it. In general, you should keep these pointers to document templates as instance data in your application's `CWinApp`-derived class. If you do, you can reference them at almost any time during the application's execution.

Each template you register with the frameworks using `CWinApp::AddDocTemplate()` is kept in a linked list. MFC uses this list to find templates when the user asks to create a new document or a new view, or performs any operation that requires that the application find an appropriate document template. For example, if more than one document template exists when `CWinApp::OnFileNew()` is called, the framework presents a list box that lets the user select the template for the type of document they wish to create.

The list is managed by an internal instance of an MFC collection class called `CDocManager`. This class is an undocumented implementation feature of MFC. Understanding how it works, though, can be quite useful. The `CWinApp`-derived object in your application creates an instance of the class. `CWinApp` holds a pointer to the `CDocManager` object. It destroys the object just before your application exits, in CWinApp's destructor function. CWinApp stores this pointer in the m_pDocManager member variable. In fact, you can use this pointer at any time to gain access to the document manager.

The document manager's main importance derives from its management of that linked list of template objects. The document manager stores the list in the public m_TemplateList member. You can walk the list using code similar to the following:

```
void CSampleWinApp::IterateEveryTemplate()
{
  CDocManager* pManager = AfxGetAp()->m_pDocManager;
  if (pManager == NULL)
    return;
  POSITION pos = pManager->GetFirstDocTemplatePosition();
  while (pos != NULL) {
    // get the next template
    CDocTemplate* pTemplate =
        pManager->GetNextDocTemplate(pos);
    // you can now do work with each pointer
    DoSomething(pTemplate);
  }
}
```

One of the most interesting things you can do with the list of templates is to drive a list of all active documents. This involves a nested loop, so that for each template you find, you can loop through the documents that the template has created. To do this, you might use some code similar to the following:

```
void CSampleWinApp::IterateEveryDocument()
{
  CDocManager* pManager = AfxGetApp()->m_pdocManager;
  if (pManager == NULL)
    return;
  POSITION posTemplate =
  pManager->GetFirstDocTemplatePosition();
  while (posTemplate != NULL) {
    // get the next template
    CDocTemplate* pTemplate =
    pManager->GetNextDocTemplate(posTemplate);
    POSITION posDoc = pTemplate->GetFirstDocPosition();
    while (posDoc != NULL) {
      CYourDocument* pThisOne = (CSampleDocument*)
          GetNextDoc(posDoc);
      // do some work with each document
      pThisOne->SomefunctionCall();
    }
  }
}
```

In both of these code fragments, you will retrieve a pointer to the manager by first getting a pointer to the application object with a call to `AfxGetApp()`. Next, you will examine the `m_pDocManager` member for the pointer to the template manager. This is, actually, more than a little duplicative, because the code fragments are member functions in `CSampleWinApp`, so they're presumably members of the very object that you are obtaining with the call to `AfxGetApp()`. Instead, you could have accessed the `m_pDocManager` member directly. However, the inefficiency lets you see the use of `AfxGetApp()` to retrieve information about the running application object. More importantly, however, you have learned how to implement the code in any function of any object in your application because the code does not presume that it is running within the `CWinApp`-derived object.

The code fragments above make use of the `CDocManager` member functions `GetFirstDocTemplatePosition()` and `GetNextDocTemplate()`, which should look familiar to you, as they perform similar processing to the `GetFirstViewPosition()` and `GetNextView()` member functions of the `CDocument` class that you learned about earlier in this chapter. `GetNextDocTemplate()` is the one that does the real work — it gets a pointer of type `POSITION` to the next document. As you can see, there is also some runtime casting in the code fragments because the second program fragment must promote the pointers to plain `CDocument` objects to pointers to the `CSampleDocument` class. It would be good programming to do `IsKindOf()` tests here, or use MFC's `DYNAMIC_DOWNCAST()` macro to make sure you get what you really wanted, but the code does not do so for simplicity's sake.

7.8.4 Destroying Documents Added with the AddDocTemplate() Member Function

If you use `AddDocTemplate()` to add your new template to the list that MFC manages for you, you need not worry about deleting the template when your application closes. However, in some circumstances, you may wish to have the template hidden from the user, and it is then that you will need to make sure your template is deleted. Deleting the template object during the program's execution of the destructor function of your application's `CWinApp` object is too good an opportunity to miss.

When designing your applications, don't worry about keeping templates around as long as you need them — as you have learned, templates are very lightweight. As with any other object, common-sense guidelines apply. A thousand templates are probably a little much (and a coding nightmare), but adding 10 or 20 templates to an application should not be overly burdensome, provided that you need the additional templates.

7.9 Understanding and Using the CView Class

As you have learned, for every `CDocument`-derived class that presents a visual interface to the user, there are one or more `CView`-derived classes that provide the interface. The `CView`-derived class provides the visual presentation of the document's data and handles user interaction through the view window.

The view window, in turn, is a child of a frame window. In an SDI application, the view window is a child of the main frame window. In MDI applications, the view window is a child of the MDI child window. In addition, the frame window can be the in-place frame window during OLE in-place editing, if your application supports OLE in-place editing. A frame window, in turn, may contain several view windows (for example, through the use of splitter windows).

7.9.1 Declaring a View Class

As earlier sections of this chapter have explained in detail, you should declare all data that is part of a document as part of the document's class. With that overriding precept in mind, however, it is important to recognize that there will likely be many data elements in your applications that pertain to a specific view. More importantly, most of those data elements will be nonpersistent, meaning you will not save them as part of the document.

Suppose, for example, that you create an application that is capable of presenting the data within the document at different zoom factors. The zoom factors will be specific to each individual view, meaning that different views may use different zoom factors even when the views are presenting information from the same document.

Given these considerations, you are probably best served to declare the zoom factor as a member variable of the view class, rather than as a variable in the document class, as shown here:

```
Class CZoomView : public CView {
  protected:
    CZoomView();
    DECLARE_DYNCREATE(CZoomView)
  public:
    CZoomableDoc* GetDocument();
    WORD m_wZoomPercent:
}
```

However, much more important than any member variables representing a setting is a member variable that represents the *current selection*. The current selection is the collection of objects within the document that the user has selected for manipulation. The nature and type of manipulation that the user might perform is entirely application-dependent, but it may include such operations as clipboard cutting and copying or OLE drag-and-drop placement support.

Arguably, the easiest way to implement a current selection is to use a collection class, just as you would in the document class. For example, you might declare the collection that represents the current selection, as shown here:

```
class CSelectableView : public CView {
  // more code here
  CList <CDocItem *, CDocItem *> m_SelectList;
  //
}
```

In addition to modifying the view class declaration, you must write one or more member functions so that your view class can respond to selection activities — filling and emptying the list, and so on. However, you must also always override the `OnDraw()` member function. The default implementation of `OnDraw()` performs no processing — you absolutely have to write code that will display your document's data items (even if the view class doesn't contain member variables of its own).

For example, if you derive your document class from `COleDocument` and use `CDocItems` to maintain the document's data, your `OnDraw()` member function for your class will probably look similar to the following:

```
void COleCapView::OnDraw(CDC *pDC)
{
  COLECapDoc *pDoc = GetDocument();
  ASSERT_VALID (pDoc);
  POSITION posDoc = pDoc->GetStartPosition();
  while (posDoc != NULL) {
    CDocItem *pObject = pDoc->GetNextItem(posDoc);
    if (pObject->IsKindOf(RUNTIME_CLASS(CNormDocItem))) {
      ((CNormDocItem *)pObject)->Draw(pDc);
    }
    else if (pObject->
        IsKindOf(RUNTIME_CLASS(COleDocItem))) {
      ((COleDocItem *)pObject)->Draw(pDc);
    }
  else
    ASSERT(FALSE);
  }
}
```

7.9.2 Analyzing the CView Member Function

Like the `CDocument` class, the `CView` class offers a wide variety of member functions that you can use in their default form and that you can override to provide specific functionality within your applications.

Among the most commonly used member functions in the `CView` class is the `GetDocument()` member function, which returns a pointer to the document object that you have previously associated with the view. Another commonly used member function is `DoPreparePrinting()`. The `DoPreparePrinting()` function displays the Print dialog and creates a printer device context based on the user's selections within the dialog. You will learn more about the `DoPreparePrinting()` function in Chapter 8.

`GetDocument()` and `DoPreparePrinting()` are the only `CView` member functions that are overridable. You can override any of the remaining `CView` member functions. These member functions supplement the large number of overridable functions that the `CWnd` class (which is the base class for the `CView` class) provides. In addition, the member functions handle the vast majority of user-interface events. Trying to list all the member functions here is a futile exercise, for there are far too many of them to make it worthwhile. However, among the member functions are functions to handle keyboard, mouse, timer, system, and other messages, clipboard and MDI events, and initialization and termination messages. Your application should override the view class member functions as needed. For example, if your application lets the user click and drag the mouse to place an object in a document, you should override the `CWind::OnLButtonDown` member function to support that functionality. In general, you can use the ClassWizard to create the override function, and simply add the appropriate code in the section the ClassWizard marks as `TODO:`

```
BOOL CSampView::IsSelected(const CObjedt* pDocItem) const
{
  return (m_SelectList.Find((CDocItem *)pDocItem) != NULL);
}
```

Another important member function that most applications will override is the `OnUpdate()` member function. During execution, the document class's `UpdateAllViews()` member function calls the `OnUpdate()` member function for each view associated with a document each time you invoke it. The default implementation of `OnUpdate()` simply invalidates the entire client area of the view window (which, in turn, results in rewarding the entire client area). To improve your application's performance, you may wish to override `OnUpdate()` and invalidate only the areas of the view window that the application must update. For example, you might implement `OnUpdate()` as shown here:

```
void CSampView::OnUpdate(CView *pView,
    LPARAM 1Hint, CObject *pObj)
{
  if (1Hint==UPDATE_OBJECT)   // app-defined constant
    InvalidateRect((CAppObject *)pObj)->m_Rect);
  else
    Invalidate();
}
```

Core Note

Normally, you should not do any drawing in the `OnUpdate()` member function. Instead, you should draw in the view's `OnDraw()` member function.

If your application supports nonstandard mapping modes such as zooming or rotating, the `CView` class `OnPrepareDC()` member function acquires special significance. In this function, you will set the view window's mapping mode before the application actually draws anything onto the window. You should always be sure, in the event that you create a device context for your view window, that your application calls `OnPrepareDC()` to ensure that the application applies the proper settings to the device context.

Similarly, your applications may often need to create a device context for the sole purpose of retrieving the current mapping of the view window. For example, you might need to convert the position of a mouse-click from physical to logical coordinates within the view's `OnLButtonDown()` member function, as shown here:

```
void CSampView::OnLButtonDown(UITN nFlags, Cpoint point)
{
  CClientDC dc(this);
  OnPrepareDc(&dc);
  dc.DPtoLP(&point);
  // further processing
}
```

7.9.3 Working with Views and Messages

In addition to those messages for which MFC provides default handlers in either `CView` or its parent class, `CWnd`, a typical view class will process many other system messages. Other messages typically include command messages that represent the user's selection of a menu item, toolbar button, or other user-interface object.

Whether it is the view or the document (or in some cases, the frame) that should handle a particular message is a decision left entirely up to you. Remember, however, that the most important criteria in making the decision is the scope and the effect of the message or command on the application's processing. If the command affects the entire document or the data stored within it, you should generally handle the command in the document class (except when the command's effect is *through* a specific view, as it might be in some implementations of a cut or paste command). If the command affects only a particular view (such as setting a zoom or rotation factor), the view object affected should handle the command.

7.9.4 MFC-Derived Variants of the CView Class

In addition to the basic `CView` class, MFC provides several derived classes that serve specific purposes, and that are intended to simplify handling of complex tasks. Table 7-1 summarizes the MFC-derived `CView` classes.

Table 7-1 MFC-Derived Variations of the CView Class

Class Name	*Description*
`CCtrlView`	Supports views that are directly based on a control (such as a tree control or edit control).
`CDaoRecordView`	Uses dialog controls to display database records.
`CEditView`	Uses an edit control to provide a multiline text editor.
`CFormView`	Displays dialog box controls. You must base `CFormView` objects on dialog templates.
`CHtmlView`	Provides a window in which the user can browse sites on the World Wide Web, as well as folders in the local file system and on a network.
`CListView`	Displays a list control.
`COleDBRecordView`	Displays database records using dialog controls.
`CRecordView`	Displays database records using dialog controls.
`CRichEditView`	Displays a rich-text edit control.
`CScrollView`	Enables the use of scrollbars for the user to move through the logical data in the document.
`CTreeView`	Displays a tree control.

Another rarely overriden variant of the `CView` class is the `CPreviewView` class. The MFC framework uses `CPreviewView` to provide print preview support to your applications.

All the `CView`-derived classes provide member functions that are specific to the class' goal. Member functions of view classes that derive from `CCtrlView` encapsulate Windows messages specific to the control class they represent.

`CFormView` and the classes that MFC derives from it (including `CDataRecordView`, `COleDBRecordView`, and `CRecordView`) support Dialog Data Exchange (DDE). You can use all four of these classes in a fashion similar to how you would use `CDialog`-derived classes, which you learned about in Chapter 4.

7.10 Understanding Splitter Windows

In a splitter window, the window is, or can be, split into two or more scrollable panes. A splitter control (or *split box*) in the window frame next to the scrollbars lets the user adjust the relative sizes of the window panes. Each pane is a view on the same document. In dynamic splitter windows, the views are generally of the same class. In static splitter windows, the views are more often of different classes. You will implement splitter windows of both kinds with the `CSplitterWnd` class.

Dynamic splitter windows let the user split a window into multiple panes at will and then scroll different panes to see different parts of the documents. The user can also unsplit the window to remove the additional views.

Static splitter windows start with the window split into multiple panes, each with a different purpose. For example, in the Visual C++ bitmap editor, the image window shows two panes side-by-side. The left-hand pane displays an actual-size image of the bitmap. The right-hand pane displays a zoomed or magnified image of the same bitmap. The panes are separated by a *splitter bar* that the user can drag to change the relative sizes of the panes.

Until now, you've learned only about applications that present one main window for their user interface. For some applications, it's interesting or valuable to have two related sections of the application's document visible in the application. Applications that can potentially render wide ranges of information to the user are common candidates for this sort of user interface. Microsoft Excel, for example, lets you split your view of a spreadsheet and independently scroll over each pane of the window (or over an entirely different portion of the sheet in each window).

Many of the applications that you will design, such as the `PaintObj` project just presented, could easily present more information than could possibly fit on one screen. Even though the application lets the user scroll within the window, the user might be interested in seeing two sections of the window simultaneously that are too far apart to ever show in a single window on a screen. By letting the user split their view of the window, you can pack more information onto the screen in the same amount of space.

Unfortunately, painting this kind of window without MFC support is tiresome, to say the least. You have to run the paint code twice, essentially fooling it into believing that the window is smaller than it really is — transposing the coordinates painted into each half of the split. Thankfully, MFC provides a simple solution: the `CSplitterWnd` class. `CSplitterWnd` is a special window class provided by MFC to live inside your application's frame window. Before you learn how to incorporate a splitter window into the design of your application, it is valuable to quickly review the different types of splitter windows that are available.

7.10.1 Differentiating Between Splitter Windows

First, programmers will generally call the `CSplitterWnd` class, and the windows it represents, splitters, so you should be aware of the different terms. Before you implement the `CSplitterWnd` class, it is worthwhile to take some time to think a little about the way a `CSplitterWnd` is used within your application, and the semantic rules that must be true for the class to make any sense and work properly.

When a user splits a window, he or she might decide to add another pane in the window either horizontally or vertically. In other words, the splitter will have to request that another view be created to fill the area to the right or below the divider. A user can also further divide a window, requiring three new views to be created immediately. This will fill the area to the right, beneath, and to the bottom right of the existing window, illustrating the quartering effect.

The `CSplitterWnd` class is capable of doing all of this work, because it records contextual information about the document template during its own creation. This lets the splitter know what document and which view class will be referenced by the new view windows. You can develop code to have the splitter generate different views for each pane in the window, or, alternatively, you can let it generate a new instance of the same view type used in the original window. You should first decide how you would like the user to approach the splitter window in your application. You will have two general choices for your splitter windows: a dynamic splitter or a static splitter.

7.10.2 Understanding Specifics of the CSplitterWnd Class

As you have learned, you will use the `CSplitterWnd` class within your MFC applications to provide users with the functionality of a splitter window, which is a window that contains multiple panes. A *pane* is usually an application-specific object that you derive from `CView`, but it can be any `CWnd` object that has the appropriate child window ID.

You will usually embed a `CSplitterWnd` object in a parent `CFrameWnd` or `CMDIChildWnd` object. Create a `CSplitterWnd` object using the following steps:

1. Embed a `CSplitterWnd` member variable in the parent frame.
2. Override the parent frame's `CFrameWnd::OnCreateClient()` member function.
3. From within the overridden `OnCreateClient()` member function, call the `Create()` or `CreateStatic()` member function of `CSplitterWnd` (depending on the splitter window type you intend to create).

Call the `Create()` member function to create a dynamic splitter window. A dynamic splitter window typically is used to create and scroll a number of individual panes, or views, of the same document. The framework automatically creates an initial pane for the splitter; then the framework creates, resizes, and disposes of additional panes as the user operates the splitter window's controls. When you call `Create()`, you specify a minimum row height and column width that determine when the panes are too small to be fully displayed. After you call `Create()`, you can adjust these minimums by calling the `SetColumnInfo()` and `SetRowInfo()` member functions.

Also, you can use the `SetColumnInfo()` and `SetRowInfo()` member functions to set an "ideal" width for a column and "ideal" height for a row. When the framework displays a splitter window, it first displays the parent frame, and then the splitter window. The framework then lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to the lower-right corner of the splitter window's client area.

To create a static splitter window, use the `CreateStatic()` member function. The user can change only the size of the panes in a static splitter window, not their number or order. You must specifically create all the static splitter's panes when you create the static splitter. Make sure you create all the panes before the parent frame's `OnCreateClient()` member function returns, or the framework will not display the window correctly.

The `CreateStatic()` member function automatically initializes a static splitter with a minimum row height and column width of 0. After you call `Create()`, adjust these minimums (just as you would with a dynamic splitter) by calling the `SetColumnInfo()` and `SetRowInfo()` member functions.

The individual panes of a static splitter often belong to different classes. A splitter window supports special scrollbars (apart from the scrollbars that panes may have). These scrollbars are children of the `CSplitterWnd` object and are shared between the two panes. You create these special scrollbars when you create the splitter window. For example, a `CSplitterWnd` that has one row, two columns, and the `WS_VSCROLL` style will display a vertical scrollbar that is shared by the two panes. When the user moves the scrollbar, `WM_VSCROLL` messages are sent to both panes. When the panes set the scrollbar position, the shared scrollbar is set.

When creating either kind of splitter window, you just specify the maximum number of rows and columns that the splitter will manage. For a static splitter, panes must be created to fill all the rows and columns. For a dynamic splitter, the framework automatically creates the first pane when the application creates the `CSplitterWnd` object.

7.10.3 Creating Dynamic Splitters

As you learned earlier in this chapter, dynamic splitters let the user split the window at his or her leisure. An application with dynamic splitters has small boxes: one above the vertical scrollbar and one to the left of the horizontal scrollbar. These can be dragged to split the window in one direction or the other. Figure 7-5 shows an application with a dynamic splitter, after the user has split the display into four window panes.

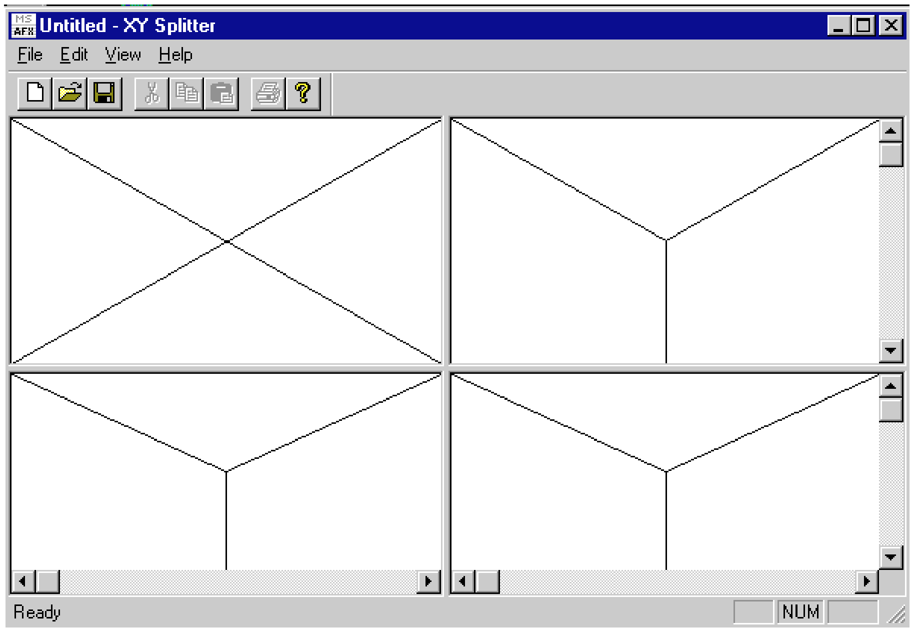


Figure 7-5 An application with dynamic splitter windows.

After dragging the box above the vertical bar down a little, the window splits and automatically creates another view: To set up this kind of splitter, you will need to declare an instance of `CSplitterWnd` in your application's frame window. For SDI applications, this would be the `CMainFrame` class, while for MDI applications, it would be within the `CMDIChildWnd` class for each view that implements dynamic splitter windows.

To initialize a dynamic splitter window, create the splitter window when the frame wants to create a client area of the frame window. Normally, the frame window will simply create the view and have it inserted into the client area of the frame, but you can have the splitter create and insert itself into the frame. The splitter will initialize a single view to populate itself, and will create more views when the user splits the window's content.

To get your frame to create the splitter, install an override of the `OnCreateClient()` function. For a dynamic splitter in an SDI application, the function just needs code similar to the following:

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
    CCreateContext* pContext)
{
  return m_wndSplitter.Create(this, 2,2,
      CSize(1,1), pContext);
}
```

The `CSplitterWnd::Create()` function accepts a few parameters. The first parameter is a pointer to the parent window of the splitter, which must be the frame. Your next two parameters are the maximum number of rows and columns that the splitter will support. You can force it to disallow horizontal splits by passing 1 for the maximum number of rows, or to avoid vertical splits by passing 1 for the maximum number of columns. Such a splitter window won't have a split box on the appropriate side of the window.

Core Note

Dynamic splitters in MFC are unable to support more than two rows and two columns. If you try to pass numbers larger than two to the `Create()` function, MFC will `ASSERT()` your debug build and not compile the application.

The value of `CSize()` that you pass to the function will cause the splitter to enforce lower size limits for the panes it creates. A size of 1 × 1, as the previous code fragment uses, effectively makes the splitter allow any window size. If, because of its content, your view has problems painting in terribly small windows, you may want to enforce a lower limit on your splitter by passing a larger `CSize()` to the creation function.

MFC won't let your user create a pane smaller than your passed `CSize()` values. It will snap the pane shut when the user lets go of the mouse while dragging a new size. Debug builds of MFC will display an appropriate warning within the debug window, such as the following:

```
Warning: split too small to create new pane.
```

Given the way all this works, with the splitter creating all of the views, there clearly must be a way for the splitter to know what view to create — and for the splitter to hook the view up to the right document. A `pContext` parameter gets passed about, from the `OnCreateClient()` parameter to the `Create()` function in `CSplitterWnd`. The `pContext` parameter points at the contextual information that tells the `CSplitterWnd` code who should handle the creation of the new view and its subsequent attachment to a document.

7.10.4 Using Different Views in Dynamic Panes

The code snippet from `CChildFrame::OnCreateClient()` shown in the previous section will result in a splitter that contains two instances of `CView`, registered in the document template that created the frame. You can use a different view in the extra panes of your splitter that lets you convey information in a different manner — side-by-side with information from the same document in a different view or even a different document in a different view.

When the user creates new panes in a dynamic splitter window, MFC calls the `CreateView()` member function of the `CSplitterWnd` class to perform the creation. Normally, `CreateView()` will simply create the required view, based on the context information you pass to it through the `pContext` parameter. If `pContext` is `NULL`, the function will determine what view is the currently active view and tries to create the same one.

You will need to derive your own class from `CSplitterWnd` if you want to have different views in the panes of your application's dynamic splitter window. You will have to override the `CreateView()` function, creating the view of your choice. Fortunately, the overriding code is simple — all you must do is pass the call along to `CSplitterWnd::CreateView()`, naming the `RUNTIME_CLASS` of the view class you wish to create for the splitter, as shown in the following code:

```
BOOL CMySplitterWnd::CreateView(int row, int col,
    CRuntimeClass* pViewClass, SIZE sizeInit,
    CCreateContext* pContext
{
  if (row == 0 && col == 0) {
    return CSplitterWnd::CreateView(row, col,
        pViewClass, sizeInit, pContext);
  }
  else {
    return CSplitterWnd::CreateView(row, col,
        RUNTIME_CLASS(CSecondView), sizeInit, pContext
  }
}
```

The code first checks to determine if the view is being created at row 0, column 0 in the splitter. If this is the case, the splitter is just now being initialized and you must create a view object of the class requested. If the code is indeed creating the first view for the splitter, it will create whatever view type the splitter originally wanted. But if the view is being created at a position other than the very first, the code will return the `RUNTIME_CLASS()` of the `CSecondView` class.

7.10.5 Using a CRuntimeClass Object

What the code in the previous section is doing is not necessarily very obvious because the calls to `CreateView()` supply a pointer to a `CRuntime-Class` object. A `CRuntimeClass` object describes the runtime type information for a class. Given this pointer, the code inside `CreateView()` can accomplish the construction of whatever object the runtime type information describes.

If you set a breakpoint on the `CMySplitterWnd::CreateView()` function and check the execution stream of an application that uses the code in the previous section, you will learn some important facts about the splitter window class. Most notably, you will find out that the splitter will destroy views that are no longer visible and recreate them later.

7.10.6 Using Splitters with Views Associated with More Than One Document

The whole process that the previous section details works fine for situations in which your new view will reference the same document as the existing views. However, if you want the second view to open another document, you have to handle the splitter's creation a bit differently. You will need to actually create the splitter and give it a different creation context. You must let it know that it must instantiate new documents and views, as well as move the view window to the correct coordinates, so that it fits with the rest of the window. Believe it or not, this last part is the most difficult portion of the process.

You can avoid doing all of this work by eliminating the call to `CSplitterWnd::CreateView()`. Instead, develop your own creation context to pass along to the `CreateView()` function, which lets it know exactly what it needs to do.

The `pContext` parameter is a pointer to a `CCreateContext` object. The `CCreateContext` object records which frame, object, and document should be used for the newly created document/view pair. The following code fragment builds its own `CCreateContext` object called `ctxSample1`. The object is initialized to have the view, document, and template information that the application should create in the new splitter panel:

```
BOOL CYourSplitterWnd::CreateView(int row, int col,
    CRun_timeClass* pViewClass, SIZE sizeInit,
    CCreateContext* pContext)
{
  CCreateContext ctxSample1;
  //if there is no active view, ASSERT
  CView* pOldView = (CView*)GetActivePane();
  ASSERT(pOldView == NULL);
  // you should test pOldView here and do something
  // reasonable with it. In this fragment, we
  // simply find out where the old view is
  ctxSample1.m_pLastView = pOldView;
  ctxSample1.m_pCurrentDoc = pOldView->GetDocument();
  ctxSample1.m_pNewDocTemplate =
  m_pCurrentDoc->GetDocTemplate()
  // pass call along
  return CSplitterWnd::CreateView(row, col,
  pOldView->GetRun_timeClass(),
  sizeInit, &ctxSample1);
}
```

7.11 Using Static Splitters

You should use static splitters in applications in which dynamic splitters are inadequate or inappropriate. Static splitters can be used when your application needs to show more than two split rows or two split columns. If you are interested in having your window split (no matter what column or row count), but refuse to allow the user to select how and where the splits should occur, you should use a static splitter instead of a dynamic splitter, because it's easier to code what you need using splitters than it is to write code to negate the actions of MFC.

Static splitters still use the `CSplitterWnd` class, but require a slightly different creation mechanism. You will still put a `CCreateWnd` instance in the `CFrameWnd` or `CMDIChildWnd` derivative of your application, but your override of the `OnCreateClient()` function will contain quite different code.

7.11.1 Creating a Static Splitter

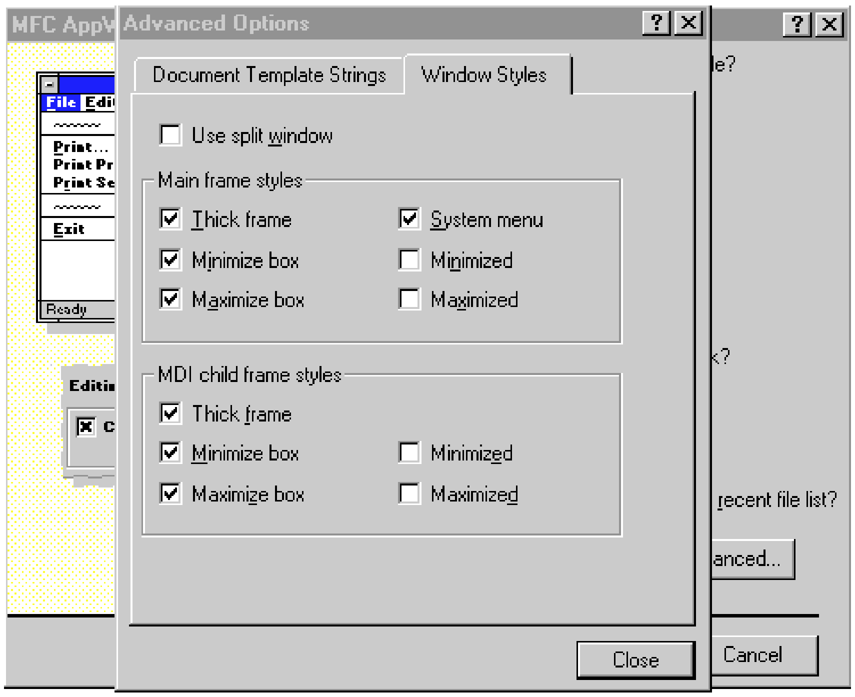
To begin with, you should call `CSpliterWnd::CreateStatic()` instead of `CSplitterWnd::Create()`. The `CreateStatic()` function still creates and wires up the splitter, but you will need to create the content for the individual panes yourself. If you do not, MFC will `ASSERT` the application and stop its execution immediately. To create the pane, call `CreateView()` on the `CSplitterWnd` object you are using. You will need to make one `CSplitterWnd` call for each splitter pane you add. For example, code to create a static splitter with five rows and three columns, would look similar to the following:

```
BOOL CMAinFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
  BOOL bRet;
  int nRow;
  int nCol;

  if(!m_wndSplitter.CreateStatic(this, 5,3))
    return FALSE;
  for (nRow = 0; nRow < 5; nRow++)
    for (nCol = 0; nCol < 3; nCol++) {
      bRet = m_wndSplitter.CreateView(nRow, nCol,
          RUN TIME_CLASS(CStaticSplitView),
          CSize(50,30), pContext;
      if (bRet == FALSE)
        return FALSE;
    }
  return_TRUE;
}
```

If you wanted to have different views in each pane, you would write the function's code to pass different `RUNTIME_CLASS()` information for each `CreateView()` call.

This chapter outlines ways to manually add a splitter window to your application mainly because a splitter window is most often an afterthought. If you are starting from scratch, you can check the Use Splitter Window in your application's MDI Child Frame or Frame Window page. You can reach this checkbox by pressing the Advanced... button in step four of the AppWizard request for information, as shown in Figure 7-6.



Advanced Options

Document Template Strings | Window Styles

- [ ] Use split window

Main frame styles

- [✓] Thick frame
- [✓] System menu
- [✓] Minimize box
- [ ] Minimized
- [✓] Maximize box
- [ ] Maximized

MDI child frame styles

- [✓] Thick frame
- [✓] Minimize box
- [ ] Minimized
- [✓] Maximize box
- [ ] Maximized

Close

Figure 7-6 Creating splitter windows within the AppWizard.

7.11.2 Understanding Shared Scrollbars

The `CSplitterWnd` class also supports shared scrollbars. These scrollbar controls are children of the `CSplitterWnd` and are shared with the different panes in the splitter. For example, in a 1 row × 2 column window, you can specify `WS_VSCROLL` when creating the `CSplitterWnd`. A special scrollbar control will be created that is shared between the two panes, as shown in Figure 7-7.

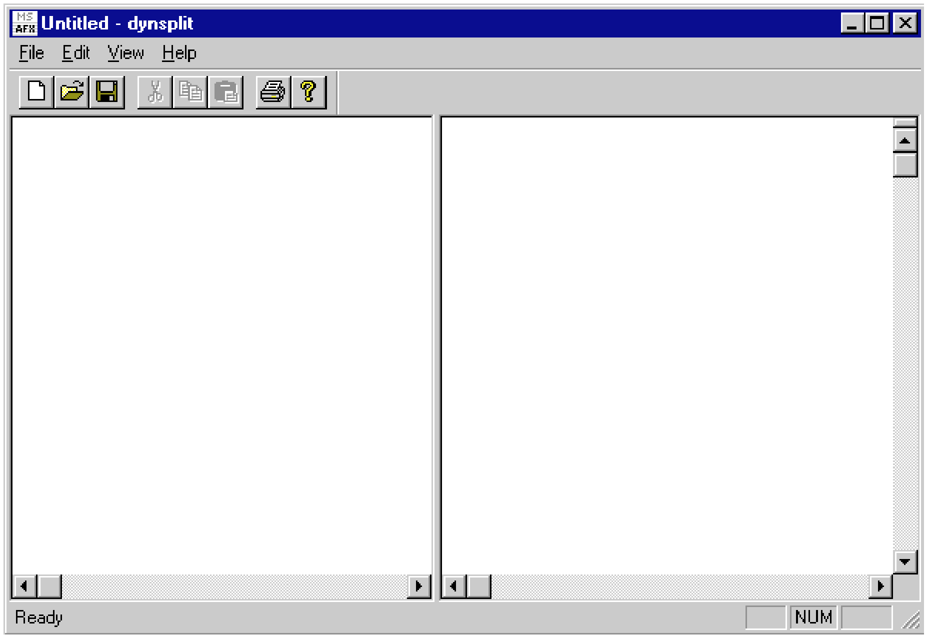


Figure 7-7 The splitter windows share a single scrollbar.

When the user moves the scrollbar, the framework will send `WM_VSCROLL` messages to both views. When the views set the scrollbar position, the shared scrollbar will be set.

Core Note

Shared scrollbars are most useful with dynamic or static splits that display two view objects of the same class within their different panes. If you mix views of different types in a splitter, then you may have to write special code to coordinate their scroll positions. Any `CView`-derived class that uses the `CWnd` scrollbar APIs will delegate to the shared scrollbar if it exists. The `CScrollView` implementation is one such example of a `CView` class that supports shared scrollbars. Non-`CView`-derived classes, classes that rely on noncontrol scrollbars, or classes that use standard Windows implementations (for example, `CEditView`) will not work with the shared scrollbar feature of `CSplitterWnd`.

7.11.3 Determining Actual and Ideal Sizes

The layout of the panes in the splitter window depends on the size of the containing frame window (which in turn resizes the `CSplitterWnd`). `CSplitterWnd` will reposition and resize the panes within the containing frame so that they fit as ideally as possible.

The row height and column width sizes set by the user, or that the application sets through the `CSplitterWnd` API calls, represent the ideal size. The actual size can be smaller than that ideal size (if there is not enough room to make that pane the ideal size) or larger than the ideal size (if that pane must be made larger to fill the leftover space on the right or bottom of the splitter window).

7.11.4 Understanding Performance Issues with Splitters

Splitters make it easy to divide the client area of your frame or MDI children to make the frame hold more than one view. However, this means that your view's painting code will be called many more times than before the split. Your view window will necessarily be smaller than it was before you adopted a splitter window, so you need to make sure your view does not do any drawing that is not absolutely necessary. Specifically, your view should not do any drawing beyond the bounds of the window. Limiting your drawing in such a manner will help ensure the greatest possible performance for your application.

Limiting the amount of redrawing that views must do is by far the most important consideration for applications that paint their views repeatedly in the different panes of a splitter window.

The likelihood that one view will change when another visible view must update its content for the same document is also much more likely when you are working with splitter windows. You should think about the different views in your application and try to ensure that your `UpdateAllViews()` or `UpdateView()` calls pass enough information to the updating view, thus ensuring that it can do the smallest amount of repainting required.

7.12 Using MFC to Subclass Windows

As you might expect, MFC provides an easy way for you to subclass any window that you derive from a `CWnd`-derived object. Rather than making the call to `SetWindowLong()`, you can simply invoke the `Subclasswindow()`

member function. Much as you would use SetWindowLong() with an API-created window, you call the Subclasswindow() member function to dynamically subclass a window and attach it to the CWnd object calling the member function. You must pass the window handle (HWND) of the window to subclass to the SubclassWindow() member function, which, in turn, will return a BOOL value that represents whether the subclassing was successful. When you dynamically subclass a window, windows messages will route through the CWnd's class first. Messages that are passed to the base class will be passed to the default message handler in the window.

On the side of the window being subclassed, the SubclassWindow() member function attaches the Windows control or window to a CWnd object and replaces the subclassed window's WndProc() and AfxWndProc() functions. The function stores the old WndProc() function in the location returned by the GetSuperWndProcAddr() member function. You must override the GetSuperWndProcAddr() member function for every unique window class to provide a place to store the old WndProc() function. You might subclass an existing edit control in a dialog box with a code similar to that shown in the following listing:

```
BOOL CSubbedDlg::OnInitDialog()
{
  ...Other initialization stuff

  // grab a pointer to the edit control
  CWnd* pEdit;
  pEdit = GetDlgItem(IDC_SSN);
  ASSERT(pEdit != NULL);

  // make the control use the system fixed-width font
  // because with numbers and dashed, it will look nicer

  HFONT hFont = (HFONT)
      ::GetStockObject(SYSTEM_FIXED_FONT);
  CFONT* pfont =:Cfont::FromHandle(hFont);
  pEdit->SetFont(pFont);

  // subclass the edit control so it is connected to the
  // custom CSubbedEdit class.
  m_Subbed.SubclassWindow(pEdit->m_hWnd);
  return_TRUE; // return TRUE unless you
               // set the focus to a control
}
```

From the point when this code is executed, the messages sent to the `IDC_SSN` control are offered first to the `m_Subbed` object, an instance of the `CSubbedEdit` class. `CSubbedEdit` is a class that subclasses (in the C++ way) from the MFC `CEdit` class. Since, after the `SubclassWindow()` call, the `CSubbedEdit` class is now an actual MFC window, it will start to receive messages via the `CCmdTarget` instance inside the `CEdit` class. You can perform the edits using ClassWizard to create message map entries for `WM_CHAR` and `WM_KEYUP` and to code whatever validation you need in response to those messages.

You can see that the `CSubbedEdit` class is a member of the application's dialog class. When the dialog initializes, the `SubclassWindow()` call is made against the social security number `(IDC_SSN)`. The program code never does anything to undo the subclassing because the functionality is disconnected when the dialog window is closed.

One additional possibility would be to use a local instance of the `CSubbedEdit` class and call `SubclassWindow()` on that. This usually isn't acceptable, since the `CSubbedEdit` instance has to outlive the control that it subclasses. Locally declaring a subclassing MFC class to a function is almost worthless because there are very few functions that continue to run while messages are being dispatched. The subclassing code would never be installed while messages were being received.

When you want to return the subclassed window to its original state, you should call the `CWnd::UnsubclassWindow()` member function to unsubclass the window. The `UnsubclassWindow()` member function returns a window handle to the newly detached window.

7.13 Alternatives to the Document/View Architecture

While the document/view model is a good default and useful in many applications, some applications need to bypass it. The point of the document/view architecture is to separate data from viewing. In most cases, this simplifies your application and reduces redundant code. As an example of when this is not the case, consider porting an application written in C for Windows. If your original code already mixes data management with data viewing, moving the code to the document/view model is harder because you must separate the two. You might prefer to leave the code as it is. There are many approaches to bypassing the document/view architecture, of which the following are only a few:

- Treat the document as an unused appendage and implement your data management code in the view class. Overhead for the document is relatively low, as described below.
- Treat both document and view as unused appendages. Put your data management and drawing code in the frame window rather than the view. This is close to the C language programming model.
- Override the parts of the MFC framework that create the document and view to eliminate creating them at all. As you have learned, the document creation process begins with a call to `CWinApp::AddDocTemplate()`. Eliminate that call from your application class's `InitInstance()` member function and, instead, create a frame window in `InitInstance()` yourself. Put your data management code in your frame window class. This is more work and requires a deeper understanding of the framework, but it frees you entirely of the document/view overhead.

7.14 Summary

Most MFC applications are based on the *document/view model*. The document, an abstract object, represents the application's data and typically corresponds to the contents of a file. The view, in turn, provides presentation of the data and accepts user-interface events. The relationship between documents and views is one-to-many: a document may (and generally will) support several associated views, but a view is always associated with exactly one document.

Your applications will derive their document classes from the MFC-provided `CDocument` class. The `CDocument` class encapsulates much of the basic functionality of a document object. In the simplest case, applications need add only member variables that represent application-specific data and provide overrides for the `OnNewDocument()` (for initialization) and `Serialize()` (for saving and loading data) member functions to obtain a fully functional document class.

More sophisticated applications will generally rely on collection classes to implement the set of objects that comprise a document. In particular, applications can use the `COleDocument` class and rely on its capability to manage a list of `CDocItem` objects that MFC does not restrict to OLE client and server objects.

You will derive view classes from the MFC `CView` base class. View windows that `CView` objects represent are child windows. In an SDI project, the parent window is the main frame window; in an MDI project, the parent window is the controlling MDI child window.

A view object, in addition to containing member variables that represent view-specific settings (such as zoom and rotation settings), often implements a current selection. The current selection is the set of document objects that the user has designated or selected within the current view for further manipulation. As with documents, most complex applications will use collection classes to manage the current selection.

While a view class may, and generally will, override many member functions of the `CView` class from which it derives, every view class must override the `OnDraw()` member function. For OLE applications, you must also override the `IsSelected()` member function. In addition, you will generally override `OnUpdate()` and `OnPrepareDC()` within your applications.

Beyond `CView` implementations that you may derive, MFC provides several derivatives of the `CView` class that you may use within your applications to handle scrolling views, views based on dialogs, controls, and views representing database records. When you design your application, you should be sure to select the class most appropriate to your application as the base class for your view class.

You've also learned about splitter windows, a powerful tool for effectively using the limited "real estate" that users will grant your applications on their desktop. Effective splitter window use lets you increase and manage not only how much information you can present to your users within your application, but how you present the information.

Finally, you learned about subclassing windows, which lets you instruct the operating system and your application as to what windows should process messages, and how to respond when the user clicks certain windows. You learned that subclassing windows, while relatively simple from the Windows API, is very easy within MFC, and that the `CWnd` class from which you will derive most windows includes the `SubclassWindow()` and `UnsubclassWindow()` member functions.

In Chapter 8, you will go one step farther with your documents and views and learn how to add printing support to your applications, how to manage the MFC printing process, and how to generate attractive output from within your applications. In the next chapter, you will work with callback functions and messages to help you better understand how the operating system communicates with your application, independent of how you implement your application's interface.