Reusing Microsoft's Foundation Class Library—A Programmer's Perspective

Wolfgang Pree
C. Doppler Laboratory for Software Engineering
Johannes Kepler University Linz, A-4040 Linz, Austria
Voice: ++43 70-2468-9432; Fax: ++43 70-2468-9430
E-mail: pree@swe.uni-linz.ac.at
http://www.swe.uni-linz.ac.at/wolf

Preview

Microsoft's Foundation Class Library[1] (MFC; Microsoft 1996) constitutes a GUI application framework implemented in C++ for the development of Windows applications. Its principal architecture builds on the lessons learned from other GUI frameworks such as MacApp (see Chapter 5 in this book), ET++ (see Chapter 7 in this book) and Interviews (see Chapter 8 in this book).

A sample application for analyzing capital gains is used to illustrate how to use MFC. A user of this capital gain analysis application enters an initial amount, the interest rate and monthly payments or withdrawals. The application plots a curve that reflects the implied effects over the years. Figure 6.1 shows a snapshot of this application.

We point out MFC's core architecture and mechanisms and demonstrate how the framework specialization is reduced by appropriate tools. A discussion of MFC's design concludes this chapter.

6.1 Features of the MFC Framework

MFC applications can perform the following functions:

- Manage an arbitrary number of windows, together with their data and the visual representation of these data.

[1] This contribution is based on MFC version 4.0 and Microsoft's Developer Studio 4.0

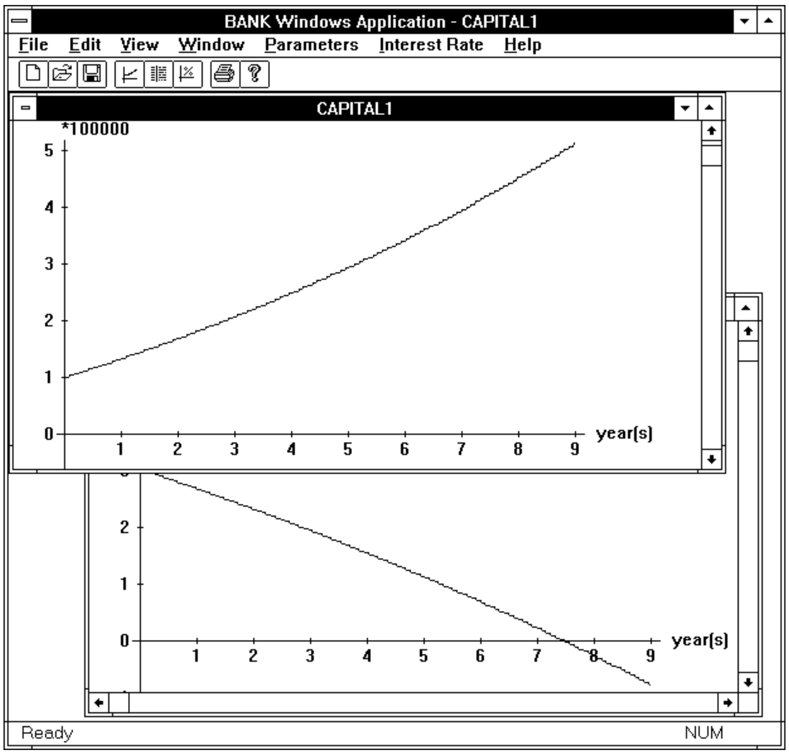

BANK Windows Application - CAPITAL1
File Edit View Window Parameters Interest Rate Help
CAPITAL1
*100000
year(s)
year(s)
Ready
NUM

Figure 6.1 Sample application built with the MFC framework

- Take care of windows (moving, resizing, activation on clicking, and so on) and their contents (invalidating regions when windows are brought in front of others, etc.).
- Handle the File menu commands Open, Save, Save As, Print and Print Preview via the corresponding default dialogs. These commands can also be invoked by pressing buttons on the tool bar.
- Process input data of control items in dialog boxes.

Supporting the following features of Windows applications requires specialization of MFC reusable components:

- Event handling: In order to make possible the selection of objects by means of the mouse, MFC provides mechanisms for handling events (mouse movements, mouse clicks, menu selection, keyboard input and so on).
- Access to relational databases: MFC database classes allow the programmer to manipulate data relying on the Open Database Connectivity (ODBC) standard. So a high degree of database management system independence is achieved.
- Object Linking and Embedding (OLE, see [Brockschmidt, 1995])
- Context-sensitive help

6.2 An MFC-Friendly Environment

The minimum set of tools required to adapt the MFC framework are a text editor and a suitable C++ compiler and linker. Fortunately, a much more powerful tool suite is offered, for example, by Microsoft's Developer Studio. Since the tools AppWizard and ClassWizard are specifically tailored to MFC, we will describe them in more detail (see Section 6.4). AppWizard and ClassWizard directly support the adaptation of MFC classes and mechanisms. AppWizard is used only once during application development, to generate all necessary classes forming a blank Windows application with default menus, window handling, etc. ClassWizard eases the task of reacting to specific events. It also allows the user to generate new classes and to map instance variables to control items in dialogs. ClassWizard is used during the whole development cycle of an application.

Furthermore, the development system smoothly integrates the following tools:

- A resource editor
- A source editor tailored to C++ sources
- A class viewer, class browser and query tool
- A component gallery
- A debugger
- A project manager
- And last but not least a C/C++ compiler and linker

6.3 MFC's Cornerstones

This section focuses on principal mechanisms and concepts incorporated in the MFC framework. The MFC framework provides elementary user interface building blocks (for example, buttons and menus), basic data structures (for example, the classes CObList, CObArray, CDate, and CString) and high-level application components such as the classes CWinApp, CDocument, CView and CWnd. Together with the elementary building blocks the high-level application components predefine, as far as possible, the look and feel of MFC-based applications.

Figure 6.2 shows the inheritance relationships of MFC classes that are relevant in this context. Abstract classes are written in italics. According to MFC's naming conventions, class names start with *C*.

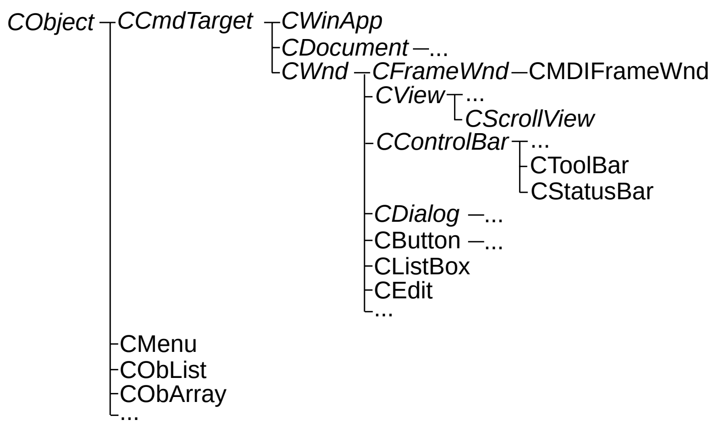


Figure 6.2 Core classes of the MFC framework

Root Class

Almost all MFC classes are derived from class CObject and so inherit its behavior. The abstract class CObject defines and partially implements the protocol for *meta-information* and object serialization.

Meta-information is a generic term for information about objects and classes. C++ does not allow inquiries about an object, such as its class. Due to this C++ deficiency, many C++ libraries implement a mechanism to provide meta-information. Like other class libraries, MFC uses macros to extract the necessary information out of a class in header and implementation files. MFC offers access to meta-information by the methods IsKindOf(*classname*) and GetRuntimeClass(). The former method returns TRUE if the receiving object is an instance of *classname* or one of its subclasses. GetRuntimeClass() return a metaclass object that can be asked for an object's size, its (base) class, etc.

Serialization by means of method Serialize allows the user to write and read the contents of an object, i.e., the values stored in the instance variables, to and from a file.

Data Structure Classes

The MFC framework offers classes to store objects in lists, arrays and dictionaries (called maps in MFC; they store key/value pairs). These components are typically used without any modifications.

Classes Defining a Generic Windows Application

In general, GUI applications are *event-driven*, and try to avoid modes. As a consequence, the user of an event-driven application ideally can enter commands in any order via input devices such as a mouse or keyboard. A GUI application has to process incoming events accordingly. For example, clicking with the mouse on the title bar of a window should be handled by a GUI application in such a way that the window can then be moved by the mouse. Clicking on the menu bar should cause the corresponding menu to open. The user should be able to select an item by moving the mouse over the menu and releasing the mouse button over the desired item. While the mouse is moved over a menu, the corresponding menu items have to be highlighted.

In MFC a CWinApp object gathers incoming events and dispatches them to the various components of the application. Each running application has exactly one CWinApp object. After a Windows application is started, MFC's WinMain function is called and sends the messages InitInstance and Run to the particular CWinApp object—actually to an instance of a subclass of the abstract class CWinApp. Invoking method Run starts the *event loop*, a loop that constantly processes incoming events (see Figure 6.3).

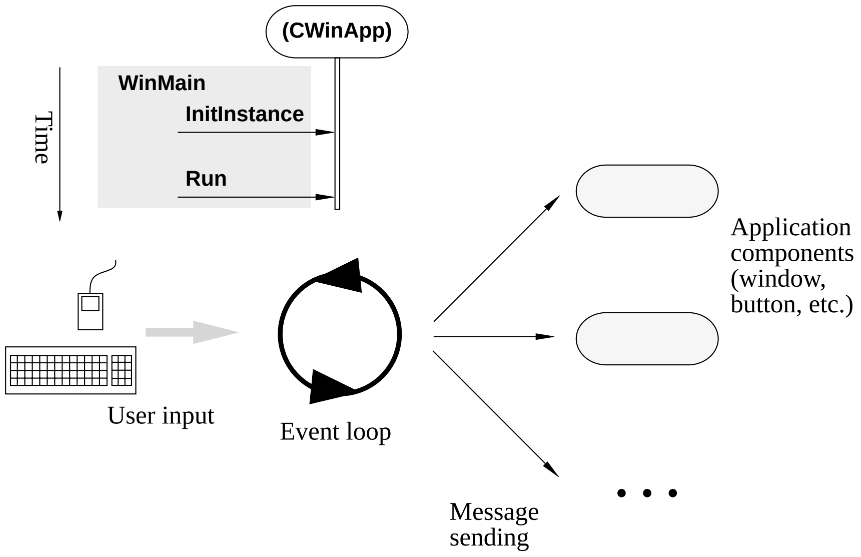


Figure 6.3 Starting the event loop in an MFC application

Many GUI applications are document-oriented; i.e., they manage documents. For example, any number of spreadsheets can be handled by a spreadsheet application. The data constituting one spreadsheet (its numbers, formulas and text contained in the cells) are handled by a subclass of the abstract class CDocument.

MFC directly supports document-oriented applications. Its core architecture is a derivate of MVC (see Chapter 2).

The MFC classes CDocument and CView correspond to the model and view components of MVC. The fact that a CDocument object can have several CView objects to display its data (i.e., model) closely resembles the MVC concept. How the controller aspect of MVC is handled in MFC is discussed below in the next section.

Another property of class CWinApp is that a CWinApp object conceptually manages any number of CDocument objects. (The actual MFC implementation deviates slightly from this conceptual view. Since this detail is irrelevant in this context we will not discuss it.)

Figure 6.4 applies the OMT notation [Rumbaugh *et al.* 1991] to depict the object structure of an MFC-based application. In the figure, the upper CDocument object refers to one CView object, through which the end user views and edits its data. The contents of the other CDocument object are displayed and edited by two CView objects. The CWinApp object takes care of all CDocument objects. For example, if the user quits the application, the CWinApp object asks the CDocument object to check whether changes in their data should be saved or not.

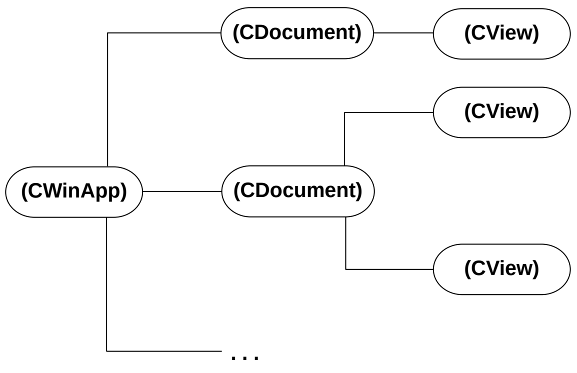


Figure 6.4 Principal object relationships in an MFC-based application

If the initial MFC adaptation is generated by AppWizard, the programmer can choose whether the application manages exactly one document (Single Document Interface, SDI) or an arbitrary number of documents (Multiple Document Interface, MDI). Class CWinApp can also be used to implement dialog-based applications such as desktop calculators.

Event Handling

The way events are handled in MFC is pretty close to the conventional C function library called Windows Software Development Kit (SDK). The SDK library supports a callback style of programming: the library functions read events and call out to various functions which the application programmer has previously registered with the library functions. These callback functions are invoked with an event identifier and parameter values, with details regarding a particular event. So applications based on SDK usually have functions with extensive switch statements determining which particular event happened. Depending on the event identifier, the additional parameter values have to be type cast. This way of handling events is error-prone—break statements might be forgotten, inappropriate type casts can cause subtile run time errors, etc.

MFC alleviates this problem a bit by encapsulating the identification of incoming events. Depending on the identified event, certain methods are called with the corresponding parameters.

Unfortunately, these event handling methods are not declared as dynamically bound methods in the MFC classes. Dynamic binding is reimplemented (for efficiency reasons?!) via clumsy macro statements. Let us illustrate this by an example. When the left mouse button is pressed over a CSampleView object (CSampleView being a subclass of CView), MFC calls method OnLButtonDown of the CSampleView object only if

- This method is overridden in CSampleView.
- The overridden method is marked as special event handling method by the add-on afx_msg in the class definition, see Figure 6.5.
- The macro DECLARE_MESSAGE_MAP is called in the class definition, see Figure 6.5.
- The macro ON_WM_LBUTTONDOWN() is contained in the message map declaration of the implementation file, see Figure 6.5.

Though this significant overhead for implementing event handling methods can be generated by the ClassWizard tool, the readability of header and implementation files suffers.

header file with class definition:

```
class CSampleView: public CView {
public:
        . . .
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        DECLARE_MESSAGE_MAP()
protected:
        . . .
};
```

implementation file:

```
. . .
BEGIN_MESSAGE_MAP(CSampleView,CView)
        //{{AFX_MSG_MAP(CSampleView)
        . . .
        ON_WM_LBUTTONDOWN()
        . . .
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
. . .
void CSampleView::OnLButtonDown(UINT nFlags, CPoint point)
{
        . . .
}
. . .
```

Figure 6.5 MFC-specific reimplementation of dynamic binding for event handling methods

If an object does not react to a Windows event MFC forwards the particular event to other objects. This forwarding mechanism is based on Windows-specific strategies which we do not discuss here.

OLE Support

In order to ease the development of OLE controls, MFC provides an object-oriented layer on top of OLE's C interface. Before presenting the integration of MFC and OLE let's first discuss the relevant concepts of OLE: The Component Object Model (COM) comprises a standard for defining the interface of objects. COM underlies OLE. COM interface descriptions cannot be changed or extended. Instead, additional interfaces can be defined for one component so that a component can offer a whole set of interfaces. Overall, COM allows interoperability between objects that are implemented in different languages and/or compiled by different compilers. Furthermore, the implementation of a component is changable without rippling its clients.

OLE 2.0 represents a library that manages the various resources (keyboard, mouse, screen) shared among components. Most available OLE components correspond to traditional applications. Usually server and client components are discerned. For example, the capital gain analysis could act as server whose documents can be referred to from an OLE client such as Microsoft Word. Word is an example of an OLE application that can act both as client and as server.

In order to enable communication between arbitrary clients and servers a minimal protocol has to be supported. In essence, a client asks its server documents to display, edit and serialize (i.e., write to an output stream) themselves. Thus if a user works with an embedded document, the client application passes control to the particular server application.

Two MFC classes have to be considered to provide an OLE client: COleClientDoc and COleClientItem. The application-specific document class has to be a subclass of COleClientDoc. COleClientItem objects represent the embedded or linked components that are managed by the particular COleClientDoc object. Most of the OLE functionality is already implemented in the two MFC classes. The programmer has to override methods such as OnInsertObject, OnEditPaste, and OnEditPasteLink in the specific View subclass. The OnChange method has to be overridden in a subclass of COleClientItem so that the displayed server document is updated.

The corresponding MFC classes to implement a server component are COle(Template)Server, COleServerDoc, and COleServerItem. Server components especially have to react to requests from clients. For this purpose several methods, which we don't discuss further, have to be overridden. For example, the framework calls the server method OnCreateDoc in case that a client application inserts a new embedded item. This method returns the appropriate OleServerDoc object.

6.4 Adaptation Support

AppWizard and ClassWizard strongly support the adaptation of the MFC framework to a specific application. In effect, an application is produced by a combination of program generation and overriding, plus some coding. Microsoft's Developer Studio integrates these and other tools into an environment.

In order to demonstrate adaptation we develop a capital gain analysis application, a part of which is shown in Figure 6.1.

Generation of the Application Skeleton

AppWizard is used once in the adaptation process. A programmer specifies various options via a dialog, for example, whether the application should handle exactly one document (SDI) or any number of documents (MDI). If the user chooses MDI, AppWizard generates the corresponding subclasses of CWinApp, CDocument, CView and CMDIFrameWnd (see Figure 6.6).

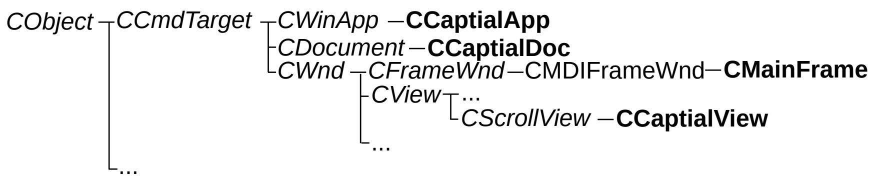


Figure 6.6 Classes generated by AppWizard (written in bold face)

Viewing a Document's Data

According to the MVC-related separation of data and their graphical representation, we add the required instance variables and access methods to CCaptialDoc: For this simple captial gain analysis, a CCapitalDoc object stores the initially invested amount, the monthly payment, the interest rate, and the date when the capital is invested. The corresponding values can be entered by the application user via a dialog box whose implementation is sketched in the next section.

In order to display the data, a grid consisting of a time axis and a captial axis as well as the capital gain curve have to be drawn. For this reason, the programmer overrides OnDraw of class CView in the subclass CCaptialView. The framework calls OnDraw to draw on the screen and the printer. MFC takes care to redraw views, for example, if windows are brought in front of others. If the data change, the document causes a redraw operation of all associated views by invoking its method UpdateAllViews.

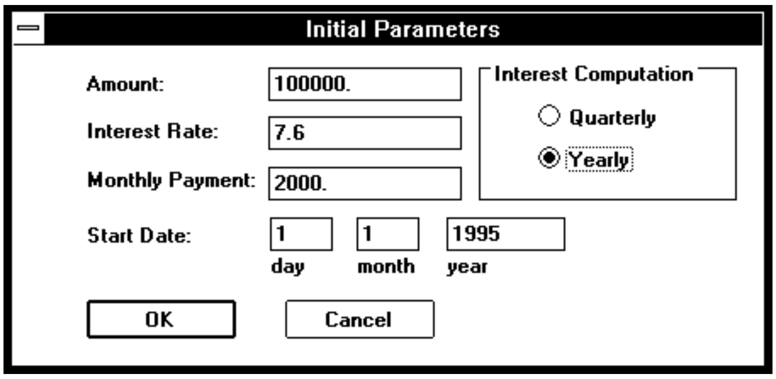
MFC encapsulates SDK-specific stuff for producing graphical output by classes such as CDC (for drawing context), CPen and CFont. Figure 6.7 shows some fragments of the OnDraw method.

```
void CCapitalView::OnDraw(CDC *pDC)
{
        CPen penStroke,
        penStroke.CreatePen(...,...,...);
        . . .
        pDC->MoveTo(CPoint(...,...));
        pDC->LineTo(CPoint(...,...));
        . . .
        // access to document’s data
        ...= GetDocument()->GetInterestRate();
                // GetDocument() is a method generated by AppWizard
                // in class CCapitalView; it returns a CCapitalDoc pointer
        . . .
        pDC->TextOut(xPos, YPos, ". . .");
        . . .
}
```

Figure 6.7 Fragments of CCapitalView's OnDraw method

Menu and Dialog Handling

This section explains how the resource editor and the ClassWizard tool automate the adaptation of the MFC framework.


Initial Parameters

Amount: 100000.

Interest Rate: 7.6

Monthly Payment: 2000.

Start Date: 1 (day) 1 (month) 1995 (year)

Interest Computation: Quarterly [✓ Yearly]

Figure 6.8 Snapshot of the Initial Parameters dialog box

Remember that a document's data should be entered via a dialog box. The application user should be able to open this dialog by choosing a menu item. We assume that the menu item Initial Parameters opens the corresponding dialog box (see Figure 6.8).

In order to implement this behavior, a programmer simply draws the menu with a resource editor. This tool works like other user interface building tools to create and rearrange items in menus, dialog boxes, and the tool bar, in a direct-manipulation interface style. A programmer defines user interface element properties in dialog boxes. Besides these item-specific properties, the resource editor assigns to each item an identifier and relates unique integer numbers to these identifiers. The identifier names can be changed by the programmer. Figure 6.9 illustrates that we chose the identifier ID_INIT_PARAMS for the Initial Parameters menu item.

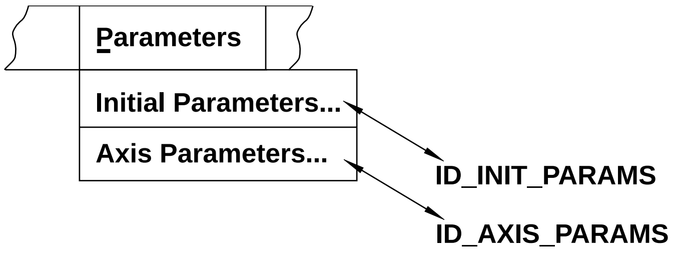
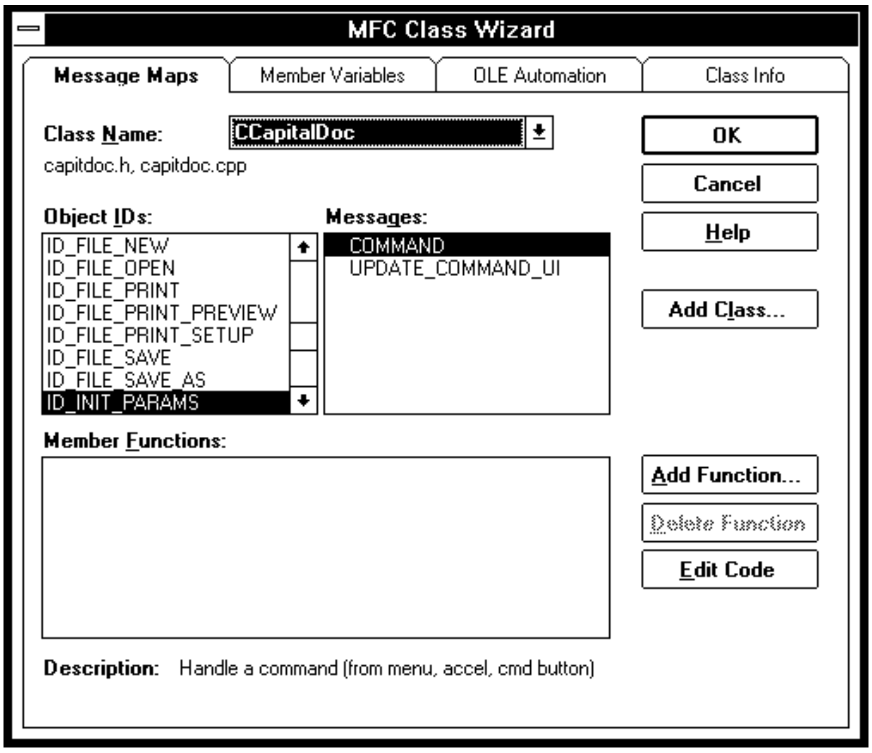


Figure 6.9 Menu specification with the resource editor

Figure 6.10 shows how to implement the handling of the Initial Parameters menu item using the ClassWizard tool.



Class Name: CCapitalDoc
capitdoc.h, capitdoc.cpp

Object IDs:
ID_FILE_NEW
ID_FILE_OPEN
ID_FILE_PRINT
ID_FILE_PRINT_PREVIEW
ID_FILE_PRINT_SETUP
ID_FILE_SAVE
ID_FILE_SAVE_AS
ID_INIT_PARAMS

Messages:
COMMAND
UPDATE_COMMAND_UI

Member Functions:

Description: Handle a command (from menu, accel, cmd button)

Figure 6.10 Snapshot of the ClassWizard tool

In the upper left combo box of ClassWizard, the class CCapitalDoc is selected as the message handler.

In connection with menu items a programmer can do the following:

- Handle the actual menu item selection.
- Enable or disable a menu item.

In order to react to a menu item selection, COMMAND has to be chosen in the right hand message list (see Figure 6.10).

By pressing the Add Function button, the necessary changes are accomplished in the header and implementation files of class CCapitalDoc analogous to the description in Section 6.3 on event handling. The programmer only has to provide the application-specific code then. Figure 6.11 lists the code generated by ClassWizard in bold face.

header file with class definition:

```
class CCapitalDoc: public CDocument {
public:
        . . .
        afx_msg void OnInitialParameters();
        DECLARE_MESSAGE_MAP()
protected:
        . . .
};
```

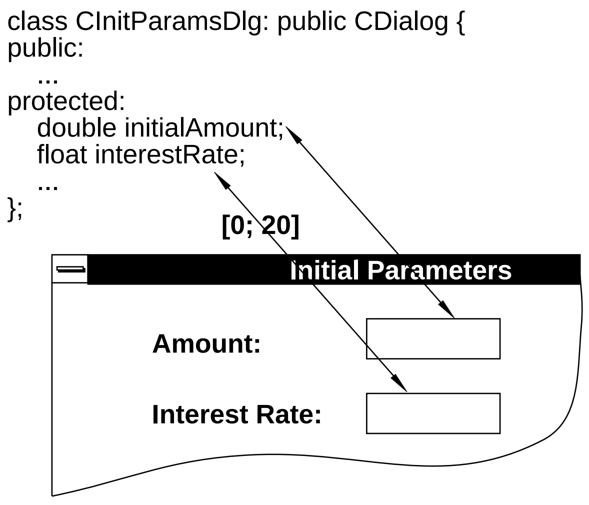
implementation file:

```
. . .
BEGIN_MESSAGE_MAP(CCapitalDoc,CDocument)
      //{{AFX_MSG_MAP(CCapitalDoc)
      . . .
      ON_COMMAND(ID_INIT_PARAMS,  OnInitialParameters)
      . . .
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
. . .
void  CCapitalDoc::OnInitialParameters()
{
      // TODO: Add your command handling code here
}
. . .
```

Figure 6.11 Code generated by ClassWizard

In order to implement OnInitialParameters, an appropriate CDialog subclass has to be implemented first. The resource editor and ClassWizard allow the programmer to generate this class. Using the resource editor, a programmer can quickly create the dialog box. Similar to menu items, dialog controls such as buttons and edit fields have unique identifiers.

The CDialog subclass that deals with the corresponding resource is created simply by invoking ClassWizard. ClassWizard automatically prompts an Add Class dialog where the programmer can change the proposed class/file names. After the CDialog subclass is generated, the programmer can add instance variables that correspond to dialog controls (see Figure 6.12).



```
class CInitParamsDlg: public CDialog {
public:
   ...
protected:
   double initialAmount;
   float interestRate;
   ...
};
```

Figure 6.12 Mapping between dialog controls and instance variables of a CDialog subclass

ClassWizard declares these instance variables in the class definition and initializes them in the constructor. Depending on the type of an instance variable, a value range can be specified, too. In this case, ClassWizard generates the necessary code in the dialog class to ensure the entering of valid data. For example, the instance variable interestRate should only accept values between 0 and 20.

The generated class CInitParamsDlg can now be used in method OnInitialParameters (see Figure 6.13). The method DoModal opens the dialog as a modal dialog (i.e., the user has to close the dialog before continuing to work with the application) and returns IDOK after the OK button was pressed.

```
void CCapitalDoc::OnInitialParameters()
{
        CInitParamsDlg initParamsDlg;
        // dialog controls <- values of document instance variables
        initParamsDlg.initAmount= initAmount;
        . . .
        if (initParamsDlg.DoModal() == IDOK) {
                // document instance variables <- values displayed
                //                                  in dialog controls
                initAmount= initParamsDlg.initAmount;
                . . .
                UpdateAllViews(...);
        }
}
```

Figure 6.13 Handling of the Initial Parameters menu item

Figure 6.14 illustrates the interaction between the principal components of the capital gain analysis application after the user has chosen the Initial Parameters menu item.

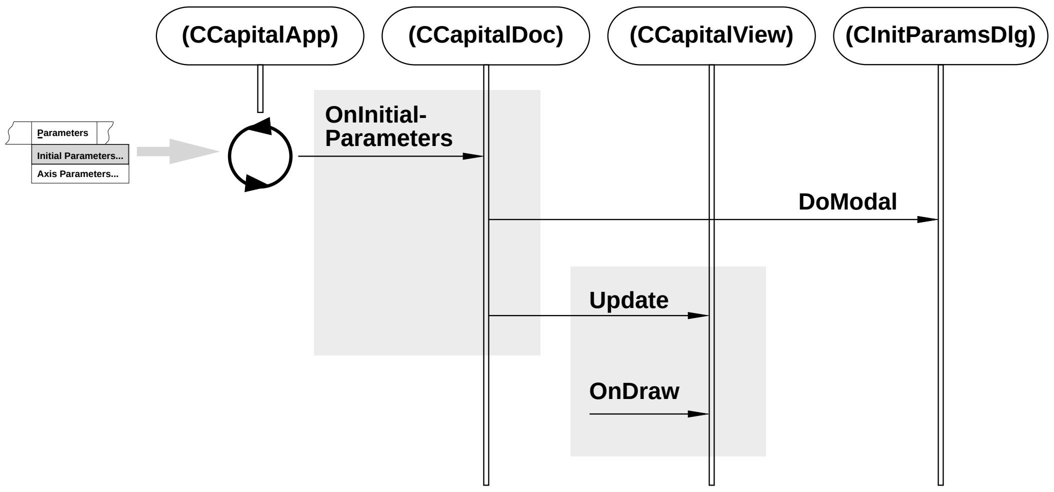


Figure 6.14 Dynamic aspects of the capital gain analysis application

6.5 Design Issues

The tool support for adapting MFC substantially eases application development. On the other hand, we recognize some short comings when taking a look at the design of the MFC framework.

Several GUI frameworks have gone through an evolutionary development process. For example, early versions of MacApp constituted a thin layer above the conventional Macintosh toolbox. Experience with a GUI framework that wraps a conventional library has corroborated that it takes time to smoothly integrate advanced abstractions of the GUI domain that significantly ease the implementation of sophisticated features (see below). MFC lacks support for features that are often essential in the GUI domain. In general, MFC is apparently a GUI framework in its early development stages for several reasons:

- MFC programmers are often confronted with SDK-related details. The MFC framework is still a thin layer on top of SDK. For example, MFC programmers have to know Windows message identifiers in the realm of event handling. Another example is that programmers deal with the Windows-SDK-specific device context when coding graphical output. Though SDK-pitfalls are avoided, MFC programmers have to be familiar with SDK-specific mechanisms. (This can also be advertised as an advantage: SDK programmers can transfer their know-how when using MFC.)
- Besides numerous spots where the MFC programmer strongly feels the Windows SDK pulse there are various features that are essential in the GUI domain but not adequately addressed by MFC:
    - MFC does not incorporate a mini-framework for undoable commands. Each MFC-based application has to implement this aspect on its own.
    - MFC does not support the implementation of direct-manipulation features (e.g., rubber-band feedback in the case of mouse tracking).
    - Scrolling/splitting of the window contents still requires too much implementation effort compared to state-of-the-art GUI frameworks (such as ET++ and MacApp). Zooming is not supported at all in MFC.

Due to these deficiencies, enormous efforts would, for example, be attached to the implemention of an MFC-based hypertext system which allows a user to edit hypertext documents (including a text drag-and-drop functionality) and to edit the graph that shows how the hypertext documents are linked.

A reason for the last two deficiencies might be that the abstraction CWnd is inappropriate. CWnd represents too heavyweight an abstraction for visual objects. All subclasses, such as control items, inherit, for example, the clipping property. Thus, it would be inefficient to base other components—such as a collection view that displays a list of items—on CWnd. For example, ET++ offers a lightweight abstraction called VObject instead of

CWnd, see Chapter 7. Many other ET++ components such as Menu, CollectionView, TreeView, GraphView, Scroller, Splitter, and Zoomer are based on VObjec, so that the last two deficiencies can be avoided elegantly.

As far as language-related implementation details are concerned, MFC confuses the programmer. Dynamic binding works in C++ only if objects are generated dynamically (new). Since MFC circumvents this C++ language feature in the realm of message handling and reintroduces it via macro calls, the programmer can declare variables statically and still have the dynamic binding flavor. But other (C++) dynamically bound methods won't work then because of the static variable declaration.

6.6 Summary

Despite MFC's deficiencies, it turns out to be significantly easier and less error-prone to develop Windows applications with MFC, compared to the conventional C function library SDK.

Though MFC does not directly address the development of sophisticated GUI applications (e.g., applications relying on an easy-to-use and intuitive direct-manipulation user interface style), MFC is well suited to produce applications with numerous dialogs. Many traditional commercial applications apply dialogs for entering parameters for database queries and displaying their results.

The MFC framework enormously benefits from the well integrated framework adaptation and development environment. Beginners might even be confused—what's in the framework and what does the environment?

Acknowledgement

I thank Helmut Hoffmann for proofreading this chapter and for discussions on various aspects of the presented framework.

Further Reading

Brockschmidt K. (1995) *Inside OLE* Microsoft Press

Microsoft Corporation (1996) *Visual C++ and Microsoft Foundation Class Libary Manuals*

Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W. (1991) *Object-Oriented Modeling and Design* Prentice Hall, Englewood Cliffs, New Jersey