Development of a Management Support System On the Windows Platform (I) :

Class structure of MFC and creation of user-defined classes

Hiroshi NOTO

Contents

1. Introduction

The purpose of this article is twofold: (1) to develop a management support system (MSS) by C++ on the Windows platform and (2) to examine the inside of the Windows programming by taking our MSS application as an example. In particular we aim at clarifying the class hierarchy of MFC (Microsoft Foundation Class) library[1] and understanding the class structure of our own application MSS based on MFC.

We adopt the Microsoft's Visual C++[2] on Windows as a language processor. Visual C++ incorporates the MFC library into its own processing program. MFC is the C++ class library which provides an object-oriented wrapper around the Window API (Application Programming Interface)[3]. MFC helps define the structure of an application program and handles many routine works on the application's behalf. MFC simplifies application development and enables us to fully exploit Windows architecture.

Whereas the introduction of MFC provides a full-featured and robust application development framework, it surely makes the mechanism of the Windows program invisible deep in the MFC library. In order to get a good skill of Windows programming, therefore, it is desirable and recommendable not only to make use of the MFC library but to understand the mechanism of the Windows programming, specifically that of the MFC architecture.

Recently we have developed an MSS application[4),17)] with MFC upon Windows in, Seminar I (for Juniors) and Seminar II (for Seniors) I am in charge of in the Hokusei Gakuen University. This means that we have gotten an example on our own of the Windows program from which we could expand our way of revealing the MFC-based Windows programming. From an educational perspective, it is worth studying to have a good grasp of what is going on inside MFC and how MFC handles (API) functions that Windows

supports. Understanding the internals of MFC will ultimately establish the basis for the students to figure out the problems（not a few, I guess）they encounter in their works of coding, debugging and applying their skills to other applications.

Throughout this article and articles to come we are going to present and explain the structure of our MSS and then to study the inside of the Windows programming, particularly its basic and fundamental functionalities and techniques with MFC, taking our MSS as an example of MFC-based Windows programming.

In §２ we review basic concepts of MFC upon Windows and present a run-down of classes grouped into four categories. In §３ we describe what the Management Support System（MSS）we have made looks like and how it works. We explain the class structure of our MSS and illustrate it diagrammatically in §４. The summary of the present article is stated in §５.

2. Basic Concepts of MFC upon Windows

Microsoft's MFC（Microsoft Foundation Class）library[1),5),12)] is designed for an efficient application framework using C++[2),8),13)] and object-oriented techniques which simplifies software development on Windows and establishes a foundation to build large-scale applications that use the latest Windows enhancements, such as COM（Component Object Model）and/or OLE（Object Linking and Embedding）in Microsoft's terminology. MFC comprises a large number of classes which encapsulate most of the Windows API（Application Program Interface）. The classes are grouped into four categories[5)]: General-purpose classes, Windows API classes, Application framework classes and High-level abstractions.

The general-purpose classes include such object classes as a strings-handling class, collection classes, exception classes, file classes, time classes and so on. One of the important properties of MFC is to wrap up the Windows API, hiding all the boilerplate code like window classes, dialog box classes. The Windows API classes also encapsulate certain Windows objects such as device contexts（device context classes）. The application framework classes handle the large pieces of a whole application. The message-pumping mechanism[15)] and MFC's document/view architecture[15)] are encapsulated here. The high-level abstractions include many useful features such as toolbars, splitter windows, and status lines and so on. Below we exemplify some of the important classes of each category which we elaborate in the following chapters tracking our application.

The general-purpose classes:

The general-purpose classes help glue the framework together which include *CObject* which is MFC's root class where almost all classes are derived from, the collection classes, the exception-handling classes,

dynamic strings, file classes, date and time classes and so on.

CObject*: *CArchive, CRuntime.

Collection classes: *CStringArray, CStringList.*

CFile.

Graphical Drawing Objects: *CGdiObject, CBitmap.*

Some other miscellaneous classes: *CPoint, CSize, CRect.*

CObject: The Mother of (Almost) All Classes

MFC's root class is called *CObject*. Classes derived from *CObject* inherit some useful capabilities, including run-time type identification for the derived class, serialization (the serialization mechanism[15] reads the RTTI (run-time type information) information to find a specified class and constructs a new object), diagnostic functions, and support for dynamic object creation. To supply these benefits, a *CObject* employs the services of several other MFC classes: *CArchive* handles object persistence (i.e. serialization) in conjunction with *CObject*. *CDumpContext* is used by diagnostic functions to output information about an object. *CRuntimeClass* is associated with every class derived from *CObject* and contains information about classs.

Collection classes: *CStringArray, CStringList*

A collection class manages a group of objects. All MFC's collection classes contain methods to add and delete elements from the collection, to retrieve elements from the collection, and to iterate over the entire collection.

CFile

Graphical Drawing Objects: *CGdiObject, CBitmap.*

A Windows GDI (Graphical Device Interface) object is represented by MFC library class. *CGdiObject* is the abstract base class for GDI object classes. A Windows GDI objects, therefore, is represented by C++ object of a class derived from *CGdiObject*. *CBitmap* provides an array of bits in which one or more bits correspond to each display pixel. We can use bitmaps to represent images.

Some other miscellaneous classes: *CPoint, CSize, CRect.*

Windows API Classes:

One of the most important things MFC does is to wrap up the Windows API, hiding all the grunge and boilerplate code. The Windows API classes themselves also encapsulate certain Windows objects such as device contexts. Here is a rundown of several Windows API classes.

Our Application: *CCmdTarget, CWinThread, CWinApp.*

***CWnd*:The mother of all Windows.**

CFrameWnd*: *CMDIFrameWnd, CMDIChildWnd.

CWnd* Derivatives: Dialogs and Controls: *CDialog.

***CDataExchange*: Dialog Data Exchange and Validation.**

Controls: ***CButton, CBitmapButton, CComboBox, CEdit, CListBox, CCheckListBox, CStatic COleControl.***

Menus: ***CMenu.***

CCmdTarget, CCmdUI, CWinThread, and the *CWinApp* classes

These classes are significantly relevant to our applications in Windows programming. Among others, "message mapping" is crucial and is implemented by the *CCmdTarget* and *CWnd* classes. A message map is the mechanism by which events are mapped to a class's method for handling those events. Any object that is derived from the *CCmdTarget* class can have a message map associated with it（the object）: the message map routes commands to the *CCmdTaget*-derived class. The class handles the messages using various member functions. The *CCmdUI* class provides a way for updating the state of a user-interface object, such as a menu or a checkbox control. The *CCmdUI* class lets us use messages to cause the command target object to update the menu status automatically. Before showing the menu items when a menu is clicked, MFC sends command update messages to the command targets in an application. *CWinThread* represents a thread of execution[16] within an MFC program. MFC supports two different kinds of threads: worker thread（real, preemptive Win32 threads）and user-interface thread. The user-interface thread has windows, and therefore it has its own message loop. The application must set up a message loop. In Windows, it is the application's responsibility to set up a message loop: The window procedure *WinMain( )* creates its window and then enters the message loop to process messages sent to the window, alternately pumping messages, i.e. retrieving messages and dispatching them to their respective window message handlers. Messages wait in a message queue until they are retrieved. The worker thread does not have windows, so it does not need to process messages. *CWinThread* encapsulates the message loop behavior. The functionality that composes a standard *WinMain( )* function is tucked away inside the *CWinThread* class and the libraries that make up MFC. *CWinApp*, which is derived from *CWinThread*, represents the standard Widnows application. *CWinApp* has overrideable functions we can use to initialize our application and to perform termination cleanup to suit your own needs.

CWnd: The mother of all Windows

CWnd is the class from which all the windows（such as dialog boxes, controls, and frame windows）are derived. Because *CWnd* is derived from *CCmdTarget*, it can receive and handle messages. Inside *CWnd*, we find the specific window's handles: most of the common API functions that operate on a window（such as *ShowWindow( )*, *MessageBox( )*, and *MoveWindow( )*）, as well as some new ones that are specific to MFC.

Frame Windows

The main frame window is usually the first window in an application to receive messages. *CFrameWnd* serves as the base class for a Single Document Interface（SDI）application's main window. *CMDIFrameWnd* provides a main frame window for Multiple Document Interface（MDI）applications, and *CMDIChildWnd* provides child windows for an MDI application.

CWnd Derivatives: Dialogs and Controls

CWnd serves as a breeding ground for all of Windows' visual interface objects. *CDialog* and all the controls are derived from *CWnd*. They have, therefore, all the same functionality as regular windows, as well as special functionality specific to themselves.

Dialogs

MFC's dialog boxes are supported by the *CDilog* class. Almost every Windows-based program uses a dialog window to interact with the user. A dialog is truly a window that receives messages, that can be moved and closed, and that can even accept drawing instructions in its client area. The Dialogs let us create either modal or modeless dialog boxes. With a modal dialog, such as the Open File dialog, the user cannot work elsewhere in the same application (more correctly, in the same user interface thread) until the dialog is closed. With a modeless dialog, the user can work in another window in the application while the dialog remains on the screen.

Dialog Data Exchange and Validation

MFC has provided a means of initializing dialog controls and automatically extracting the data from them when the [OK] button is pressed at the end of the dialog. Dialog data exchange and validation (DDX/DDV) is implemented through the *CDataExchange* class.

Property Sheets

Property Sheets or tabbed dialogs are useful for reducing the number of dialogs in our application by layering them all in one dialog. MFC supports property sheets through two classes: *CPropertySheet* and *CPropertyPage.*

Controls

MFC wraps each of the basic Windows controls into their own classes. All controls are derived from the *CWnd* object. The *COleControl* class is a powerful base class for developing OLE (Object Linking and Embedding) controls derived from *CWnd*. An OLE control is an embeddable OLE object that has additional support for OLE controls interfaces, among which included are required interfaces, optional methods, properties, property pages, ambient properties, automation methods, event firing, and self registration.

Menus

Windows menus are handled by the *CMenu* class. All the menuing functionality is encapsulated within *CMenu,* including the menu handle and methods to load, update, track, and destroy a menu.

Application Framework Classes:

An application is a superset of a class library. An ordinary library is an isolated set of classes designed to be incorporated into any program, but an application framework defines the structure of the program itself and offers all that is needed for a generic application. The application framework classes handle the large

pieces of a whole application. The message-pumping logic and MFC's document/view architecture are encapsulated here.

Document/View Architecture: *CDocTemplate, CSingleDocTemplate, CMultiDocTemplate, CDocument, CView.*

Document/View Architecture

The idea behind the document/view architecture is to separate actual data from representations of that data. In other words, for one set of data, you can generate multiple views of it. MFC provides several classes that implement the document/view architecture. These include the following:

CDocTemplate, CSingleDocTemplate, CMultiDocTemplate

The document template is the glue that holds together a document and its views. It is the liaison between documents, views, and frame windows. A *CSingleDocTemplate* supports one document at a time, whereas a *CMultiDocTemplate* supports multiple documents simultaneously. Both are derived from the *CDocTemplate* class.

CDocument

The *CDocument* class handles the data within our application. Whatever we open using the File/Open is usually our document. When we derive a class from *CDocument*, we add variables to hold our data, to code for reading and modifying our data, and to code for writing our data back out to disk.

CView

A *CView* represents the actual window we see on our screen. *CView*, which is derived from *CWnd*, is what we use to show our data to the outside world. The view renders our data on an output device such as a screen or a printer. MFC virtualizes the rendering in such a way that our *CView*'s *OnDraw( )* method is called whenever the view must be rendered and that the code which draws our data on the screen is the same code that renders our image on the printer. It simply uses a different device context. The device context is the key Graphical Device Interface（GDI）element that represents a physical device. Each device object has an associated Windows device context. The view is also a command target, so it can receive your input as well.

High-level abstractions:

MFC provides a number of enhancements to the *CView* class, one of which is *CFormView*.

Form View: *CFormView.*

A *CFormView* is a scroll view melded with a dialog template. As with a regular dialog box we can plant various controls（edit boxes, list boxes, radio buttons, and so on）. Because the *CFormView* is derived from a *CScrollView*, it has all the other inherent capabilities of a *CScrollView*, including scrolling, command handling, and document management through a document template.

3. Design and Performance of Management Support System (MSS)

The MSS system[4),17)] is made up of two subsystems: one is DSS (Decision Support System) and the other is ES (Expert System). DSS presents a simulation framework (or generator) based on a specific model which is implemented from outside. The model requires one or more scenarios (usually represented by (a) set(s) of variables with specified values. We adopt here a "finance model"[17)]. The simulation framework realizes "goal-seek" algorithm such that with a set-up of a specific goal at the final term, the framework calculates the condition in the initial term necessary to achieve or meet the goal. Executing the simulation model, we finally obtain the values of all the variables characterizing the scenario. Next the simulation model output is fed into the ES system. The ES evaluates the input data through the expertise expressed by if-then rules of the system program. In other words the ES makes reasoning (i.e. theorem proving inference called "resolution refutation by backward reasoning"[18),19)]) based on the rules and the facts (the latter the simulation output from the DSS), whether the simulation results satisfy a set of if-then rules. In the present case of a specific problem domain, the ES can render advice on and recommendation for the financial status of an individual or a business organization.

We briefly review what our MSS looks like and how the MSS behaves: (1) We look at the layout of the windows, the contents of the menus, of the dialog boxes and of the property sheets and pages. (2)We outline the functions assigned to the menus in the main window and to the controls in the dialog boxes. (3)We list and look over the input and output files which are transferred between the controls and the storage areas according to the execution of the program.

In MSS we use eight files of which the names and the contents are as follows.

"vc.d": contains the names of variables and commons.*)

(*)In general, variables are grouped into one or more bunches of them called "commons" which are assigned to their respective models. The MSS requires a specific "model" to carry out its framework.

"cm.d": describes the relation between "common" and "model".

"csf.d": contains a file name(a complete name including its path) which describes "common" name and "scenario number".**)

(**)One "scenario number" represents one particular set of values of the variables belonging to the "common" under consideration. There could be two scenarios or more for one "common".

"cfin1.dat": expresses a set of variables with specified values characterizing common "cfin" and scenario number "1".

"report.txt": a text file used for remembering intermediate and/or final results and for taking notes technical or conceptual, about the present simulation and evaluation by executing the MSS.

"cfin1.pro": the same as "cfin1.dat" in contents. "cfin1.pro", however, comprises part of the Expert system source codes in PROLOG[14] with its extension "pro".

"analysis.pro": the whole source program written in PROLOG of our Expert system.

"MSSsystem.bmp": a bit map file used as a splash screen that pops up when our MSS first starts up.

The MSS application exploits the MDI（Multi Document Interface）model which supports two or more documents to be opened at the same time. For the application's "view" we use *CFormView* as the base class. The *CFormView* class is associated with dialog templates that implement the frame（i.e., window）characteristics and enumerate the controls（i.e., menus, buttons, edit boxes, and so on）.

We prepare five document templates which enable us to store and manipulate four document files with one view each and one document file with two views. The document template class（*CDocTemplate*）establishes the relationships among the four classes: the framework class, the document class, the view window class and the MDI child frame window class. The document template is a "road map" used by the framework and has pointers to the document, view and child frame classes. The framework instantiates those classes to create a new child window.

Let us start up the MSS application. A splash screen appears immediately followed by a pop-up dialog box menu such as the one shown in Fig. 2-1. Selecting "Describe" document template object opens its child frame window titled "Describe 1" inside the main frame window titled "MSS"（Fig. 2-2）. The main window contains the menu bar on which we can see an item "Describe" in addition to familiar items such as "File" and "Edit"... A click on "Describe" triggers a drop-down submenu like in Fig. 2-3. Selecting item "common→va riable" opens another child frame window titled "Vc.d" and executes the assigned command menu handler. This handler reads "vc.d" file and deals with what the submenu requires: in the edit box "common→variabl e", the common name "cfin" in the present case is shown and all the 15 variables belonging to it are displayed（Fig.2-4）. Almost the same processes go on as for the remaining two submenu items: the items "variable→ model" and "common and scenario number→data file name" read "cm.d" and "csf.d" files, respectively. When we select item "variable→model" a dialog box "Variable and Model" pops up（Fig. 2-5-1）. We enter some variable name in the edit box to find out which model the variable belongs to. A click on［OK］button opens the third child frame window titled "Cm.d" and starts up a handler searching the array which is created from "vc.d" file（Fig. 2-5-2）. If an element in the array matches the variable name, the handler this time accesses "cm.d" file with "common" key('cfin') which is known from "vc.d" and finally returns "model" name that the variable in question belongs to, in the edit box in the child window（Fig.2-5-2）. We can create type-safe arrays for the data of any type by using the generic *CArray* class. Here if we enter a wrong variable name, say 'depric', then the handler returns an error message（Fig.2-5-3）.

Likewise selecting item "common and scenario number→data file name" opens the fourth child frame

window titled "Csf.d" and displays in the edit box the "common" name, the "scenario number" and the complete file name of "Csf.d" including its full path (Fig. 2-6).

Now we use the second document template object by selecting [File/New] menu commands on the toolbar in the main window. Immediately the standard MFC dialog box pops up again that prompts the user to select a document-template object. Selecting "Run" document template object opens the fifth child window "Run1". The menu commands [Run/Execute] on the menu bar in the mainframe window display a pop-up dialog box(Fig. 2-7-1). The "Execute" command is just about to read the file "Cfin1.dat" and carry out the simulation with the "Finance Model". Specifying the values for the required items, such as "model name", "initial time". "final time" and "scenario number", we click on the [OK] button, which brings about the sixth child window titled "Cfin1". In the child window "Cfin1" we can see the results of our simulation where looping three times ('.Time' stamped each time) from the initial time('1') to the final time('3'), the program calculates all the variables in the Cfin1.dat with lists of the items before simulation in the edit box and after simulation in juxtaposition in addition to the list of required condition for 'Run' execution (Fig. 2-7-2). Numbers in many figures are read in dollar. The document template "Run" is simply for checking the looping procedure through the Cfin1. dat as just described above.

Next we select "Goal-seek" document template object. The "Goal-seek" item on the main bar has four items as the submenu, "Exogenous input variable's dependence on term", "decision variable and iteration", "execution" and "graphic data generation" (Fig. 2-8-1). The item "Exogenous input variable's dependence on term" in the drop-down menu owns a cascade menu which has two items "Fixed Rate" and "Fixed Value". Here we adopt "Fixed Rate" determination of the dependence of a exogenous input variable on the unit time (term or period) (Fig. 2-8-2). The item "decision variable and iteration" triggers a property sheet titled "decision variable and iteration". The property sheet enables us to insert a page of controls in a tabbed dialog box. We can switch from one page to another with a mouse click on the tab. In the present case we have two pages of which the names appear on the tab at the top of the property sheet: "Alternative" and "Setup Iteration". In "Alternative" page, we select two decision variables with their time dependence assumed to find out an alternative course of action (Fig. 2-8-3). In "Setup Iteration" page we refine iteration performance, assigning the values of the following quantities; "tolerance", "maximum times of iteration", "delta", and "error" (Fig. 2-8-4). The "goal-seek" operation invokes Newton's method as its simplest case to search for a numerical solution to the equation. The Newton function evaluates the result to see whether it finds a solution to within the accuracy specified by "tolerance". The item "maximum times of iteration" designates the maximum number of steps to take in trying to find a solution. "delta" represents a small interval of increment of the independent variable in each iteration step. We put a starting value into "error" which is defined as the difference between the goal and the obtained value in the i-th step (at first i = 0). After setting all the conditions for "goal-seek" iteration on each page we click on [OK] button. Then we can select "execute" in the submenu of the "Goal-

seek" menu on the toolbar in the main window. Selecting "execute" triggers a dialog box which specifies the conditions for our goal-seeking analysis such that it calculates the amount of an "exogenous input variable" at the initial unit time to achieve a desired level of an "output variable" at the final unit time（Fig. 2-8-5）. Immediately after pushing［OK］button, we get in the seventh child window "Cfin1" as shown in Fig. 2-8-6, the result of the goal-seek analysis based on the "Finance Model".

We combine the two systems（or modules）, DSS（Decision Support System）and ES（Expert System）to make the present MSS（Management Support System）. We are now in a position to consult the facts the DSS outputs and perform inference invoking the rules stored in ES. We select here "Reasoning" document template object which opens the eighth child window titled "Reasoning1"（Fig. 2-9-1）. The "Reasoning" item on the main bar has four items as the submenu. "Read Cfin1 data", "Insert Cfin1 to Analysis.pro", "Execute Expert System" and "Save 'Report'". The item "Read Cfin1 data" on the drop-down menu reads "Report.txt" file. At the same time this submenu reads the facts derived by the "Goal-seek" simulation,i.e. "Cfin1.dat" file which are shown in the "Cfin1.data" edit box（Fig. 2-9-2）. The item "Insert Cfin1 to Analysis.pro" reads both "Cfin1.pro" and "Analysis.pro" program files and "consults" the facts "Cfin1. pro"（the same as the "Cfin1.dat in contents）, i.e. the submenu item directs the facts from the DSS into the ES（"Analysis.pro"）for reasoning as shown in the "Analysis.pro" edit box（Fig. 2-9-2）. The item "Execute Expert System" starts up our Expert system program "Analysis. pro" : Clicking on "Execute Expert System" opens a new window titled "Analysis.pro" supervised by Prolog compiler, "Strawberry Prolog"[20]. In the Prolog window the submenu item "compile" compiles "Analysis.pro" and the submenu item "run" starts the program in the active window. The member function pursues inferences based on the rules and the facts. The messages（i.e. the results for the reasoning）from the ES are sent back to the edit box in the Prolog window as shown in Fig. 2-9-3. The "Save 'Report'" saves the "Report.txt" file which could be used as a memo file which contains all the necessary output results and the remarks for future use.

Finally we select once again "Goal-seek" document template object. The document template is the same as the former "Goal-seek" document template and opens the nineth child frame window "Goalseek2". However this time we click on the "graphic data creation" submenu item brought up from the "Goal-seek" item on the main bar. The dialog box "Conditions of Drawing Graph" appears for various setups in order to draw a line chart: "data point number" and "partition number", "maximum value and minimum value on the x-axis scale", "maximum value and minimum value on the y-axis scale" and "number of grid lines for horizontal lines and vertical lines"（Fig. 2-10-1）. Clicking on［OK］butotn, we show in Fig. 2-10-2 a graph plotting the goal-seek simulation results, with the abscissa specifying the output variable and the ordinate specifying the exogenous input variable. The desired value of the goal we set up is shown with red line also in Fig. 2-10-2. We utilize an ActiveX control to make the program for drawing a line chart and register it as an OCX file in the registry. An ActiveX control is essentially a simple OLE（Object Linking and Embedding）object［a kind of program］

without having to support particular interfaces to qualify as a control. We can add an ActiveX control in the dialog box like a common control if the ActiveX control is registered as the OCX file in the registry.

4. Class Structure of MSS

In this chapter we present the class hierarchy chart pertaining to the MSS (Management Support System). In Fig. 4 we display most of the classes of the MFC (Microsoft Foundation Class) library relevant to our MSS, where we have added twenty four "our 'user-defined classes'" (which are shown as double squares) to construct the basic structure of our application program.

Most of the classes in the Microsoft Foundation Class Library are derived from a single base class at the root (the root class, *CObject*) of the class hierarchy. *CObject* provides a number of useful capabilities to all classes derived from it. Those classes can be divided into the following categories:

Application Architecture

File Services

Graphical Drawing

Graphical Drawing Objects

Arrays

Lists

Simple Value Types

Window Support

Frame Windows

Dialog Boxes

Views

Controls

User Windows Support

In the "Application Architecture" category, we derive "*CMSSApp*" class with the *CWinApp* class as its base class. "*CMSSApp*" provides member functions for initializing our application (and each instance of it) and for running the application. Each application that uses the Microsoft Foundation classes can only contain one object (in this case "*CMSSApp*") derived from *CWinApp*. As for "Document" classes we derive four Document classes, *CMSSDoc*, *CMSS1Doc*, *CMSS2Doc* and *CMSS4Doc* from the base class *CDocument*. The *CDocument* class provides the basic functionality for "user-defined" document classes. It supports standard operations such as creating a document, loading it, and saving it. "*CMSSDoc*" deals with the files "vc.d", "cm.d" and "csf.d". Likewise "*CMSS1Doc*" "cfin1.dat" and "*CMSS4Doc*" "Report.txt". "*CMSS2Doc*" handles the files "cfin1.pro" and "Report.txt".

In the Graphical Drawing Object Category we derive "*CWzdBitmap*" class from the *CGdiObject* class. The *CGdiObject* class provides a base class for various kinds of Windows graphics device interface（GDI）objects such as bitmaps, regions, brushes, pens, palettes. We draw a splash screen bitmap for our application in an instance of "*CWzdBitmap*".

In the Frame Windows category we have the *CMDIFrameWnd* class and the *CMDIChildWnd* class. Both are derived from the *CFrameWnd* class which serves for a Windows single document interface（SDI）overlapped or pop-up frame window. We derive "*CMainFrame*" from the *CMDIFrameWnd* class for the functionality of a Windows multiple document interface（MDI）frame window. Similarly we derive "*CChildFrame*" from the *CMDIChildWnd* class for the functionality of a MDI child frame window. Objects of class *CPropertySheet* represent property sheets, otherwise known as tab dialog boxes. A property sheet is displayed by the framework as a window with a set of tab indices, with which the user selects the current page and an area for the currently selected page. We make "*CSetupAlternative*" class derived from the base class *CPropertySheet* for preparing two *CPropertyPage* classes（see the next paragraph）.

In the Dialog box category we derive six "user-defined" classes from the base *CDialog* class: *CAboutDlg*, *CVarDialog*, *CSelectDlg*, *CFixedRateDlg*, *CSetGoalDlg* and *CDataGenDlg*. The *CAboutDlg* class displays the information on the application's name with its version number and a copyright notice. The *CVarDialog* class is responsible for a dialog box which pops up when selecting the menu ["Describe"/ "variable→model "] in the child frame window "Describe1". Likewise the *CSelectDlg* class is responsible for a popped-up dialog box when selecting the menu ["Run"/"Execute"] in the child frame window "Run1". The *CFixedRateDlg* class manages a dialog box which pops up when selecting the menu ["Goalseek"/"Exogenous input variable's dependence on term"/"Fixed Rate"] in the child frame window "Goalseek1". The *CSetGoalDlg* class handles a popped-up dialog box when selecting the menu ["Goalseek"/"Execute"] in the child frame window "Goalseek1". The *CDataGenDlg* class is responsible for a dialog box which pops up when selecting the menu ["Goalseek"/"graphic data creation"] in the child frame window "Goalseek2". The *CPropertyPage* class is derived from *CDialog*. We have derived two classes from the base class *CPropertyPage*: *CAlternative* and *CIteration*. The class *CAlternative* defines the attribute of the dialog box with a tab name "Alternative" which pops up when selecting the menu ["Goalseek"/"decision variable and iteration"] in the child frame window "Goalseek1". Likewise the class *CIteration* defines the attribute of the dialog box with a tab name "Setup Iteration" which pops up when selecting the menu ["Goalseek"/"decision variable and iteration"] in the child frame window "Goalseek1".

As for View category we derive five classes from the base *CView*: *CMSSView*, *CMSS1View*, *CMSS2View*, *CMSS3View* and *CMSS4View*. The *CView* class provides the basic functionality for "user-defined" view classes. A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen or printer and interprets user input as

operations upon the document. *CMSSView* responsible for the child frame window "Describe1". Likewise *CMSS1View* for "Run1", *CMSS2View* for "Goalseek1", *CMSS3View* for "Goalseek2" and *CMSS4View* for "Reasoning1". In the Control category we construct "*CActiveXCon_LChartCtrl*" class from the base class "*COleControl*" which is a powerful base class for developing OLE controls. Derived from *CWnd*, this class inherits all the functionality of a Windows window object plus additional functionality specific to OLE (Object Linking and Embedding) such as event firing. The "*CActiveXCon_LChartCtrl*" class defines ActiveX control for drawing a line chart in the child window "Goalseek2". Finally we derive "*CWzdSplash*" class from the base calss *CWnd*, which is responsible for creating a splash screen to display our seminar "logo" along with a background image.

5. Summary

In the present article we have presented the management support system (MSS) developed in the 'Seminar I' and 'Seminar II' I am in charge of in the Management and Information Department of Hokusei Gakuen University. The MSS is written in the Visual C++ language processor which basically incorporates MFC class library. MFC defines the structure of our present application and handles many routine works on the application's behalf. In order to understand the structure of the MSS it is crucial to comprehend the MSS class structure itself, in particular its basic and fundamental functionalities and techniques with MFC: first we describe the basic concepts of MFC and then give a run-down of classes grouped into four underlying categories. Prior to elaborating the class structure of our MSS we explain the performance of the MSS, i.e. what the MSS main frame window and child frame windows look like and how the MSS behaves and works responding to our operation, such as handling the menus, the pop-up dialog boxes and/or property pages and clicking on the controls and so on. Then we depict and illustrate the relevant twenty four "user-defined classes" in our MSS which are derived from their respective base classes, each functionally grouped into categories according to the class chart of the MFC class library. We find that the MSS is realized basically exploiting the class structure of MFC of which the functionalities are inherited to our "user-defined" classes and which makes the Windows programming much more efficient, transparent and rewarding.

[Acknowledgments]

The present interface programs are developed by using Microsoft Visual C++ ver. 6.0 under Windows2000 and WindowsXP. The PROLOG program is written in and processed by Strawberry PROLOG ver. 2.3 developed by D.D.Dobrev, Sofia, Bulgaria. Strawberry PROLOG performs under Windows2000 and WindowsXP.

[References]

1) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_class_library_reference_introduction.asp

2) http://msdn.microsoft.com/library/default.asp?URL=/library/devprods/vs6/visualc/vctutor/tutorhm.htm

3) An API encompasses all the function calls that an application program can make of an operating system, as well as definitions of associated data types and structures. The API also implies a particular program architecture.

4) Noto, Hirosi. Interface Reflecting a Hybrid of the Decision Support System and the Expert System, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 37, 2000.

5) George Shepherd and Scot Wingo. MFC Internals, Addison Wesley Developers Press, 1999.

6) Aran R. Feuer. MFC Programming, Addison Wesley, 1997.

7) Eugene Kain. The MFC Answer Book, Addison Wesley, 1998.

8) John E. Swanke. Visual C++ MFC Programming by Example, R&D Books, 1999.

9) David J. Kruglinski, George Shepherd, and Scot Wingo. Programming Visual C++ (fifth edition), Microsoft Press, 1998.

10) Stephen D. Gilbert and Bill McCarty. Visual C++ 6 Programming Blue Book, CORIOLIS, 1999.

11) Hayasi, Haruhiko. A New Introduction to Visual C++ Ver. 5.0 (Beginners edition)(in Japanese), SoftBank Books, 1998.

12) Yosida, Kouitirou. Kiwameru Visual C++ (in Japanese), Gijutu Hyouron-sya, 1998.

13) Yamasita, Hirosi; Kuroba, Hiroaki and Kuroiwa, Kentarou. C++ Programming Style (in Japanese), Ohmsha, 1994.

14) Ivan Bratko. PROLOG Programming for Artificial Intelligence, Addison-Wesley,1986.

15) The architecture will be discussed in detail in the succeeding articles.

16) A process is a running program that owns its own memory, file handlers, and other system resources. An individual process can contain separate execution paths, called threads. Even a single-threaded application has one thread, the main thread.

17) Iijima, Jun'iti. Decision Support System and Expert System (in Japanese), Nikka Giren, 1993.

18) Peter Jackson. Introduction to Expert Systems (third edition), Addison-Wesley, 1999.

19) Efraim Turban and Jay E. Aronson. Decision Support Systems and Intelligent Systems (fifth edition), Prentice Hall, 1998.

20) D.D.Dobrev. Strawberry PROLOG ver. 2.3, Sofia, Bulgaria, 2002.

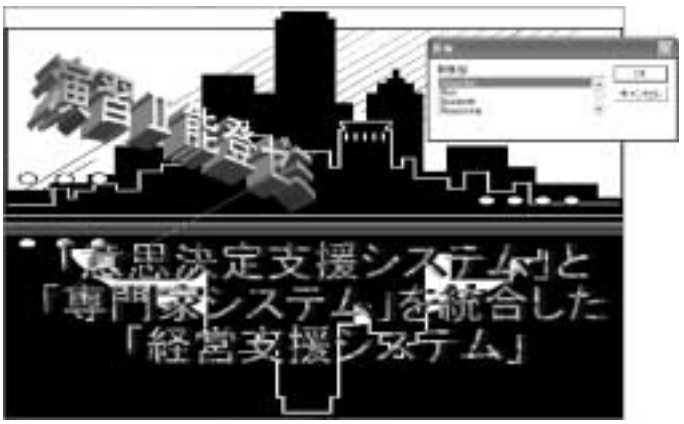

Fig. 2-1.
A splash screen immediately followed by a pop-up dialog box menu.

Fig. 2-2.
A child frame window "Describe1" inside the main frame window "MSS".

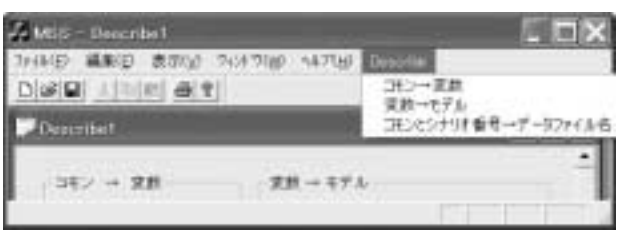

Fig. 2-3.
A click on "Describe" triggers a drop-down submenu.

Fig. 2-4.
All the 15 variables belonging to the common name "cfin" in the edit box "common → variable" in a child frame window "Vc.d".

Fig. 2-5-1.
Entering variable 'deprec' in the edit box in the dialog box "Variable→Model".

Fig. 2-5-2.
The "model" name 'finance' is looked up and displayed for variable 'deprec' in the edit box "Variable→Model" in the child frame window "Cm.d".

Fig. 2-5-3.
An error message is shown for a wrong variable name 'depric'.

Fig. 2-6.
Here displayed are the "common" name, the "scenario" number and the complete file name of "Csf.d" file in a child frame window titled "Csf.d" in the edit box "common and scenario number→data file name".

Fig. 2-7-1.
A pop-up dialog box "Specifying items for 'Run'" for four variables with their values given.

Fig. 2-7-2.
The results of the simulation with the "Finance Model". Left: a list of the required condition for 'Run' execution. Middle: a list of all the items before simulation. Right: a list of all the items after simulation, where '. Time' is stamped for each loop.

Fig. 2-8-1.
The "Goal-seek" item on the main bar has four items as the submenu. The first item "Exogenous input variable's dependence on term" in the submenu owns a cascade menu which has two items.

Fig. 2-8-2.
A dialog box for "Fixed Rate" determination of the dependence of a exogenous input variable on the unit time (term or period).

Fig. 2-8-3.
"Alternative" page in "Decision variable & Iteration" property sheet specifies two decision variables with their time dependence assumed.

Fig. 2-8-4.
"Setup Iteration" page in "Decision variable & Iteration" property sheet refines iteration performance assigning the values of relevant four quantities.

Fig. 2-8-5.
A dialog box "Conditions for Goal-seek Simulation" which specifies the conditions for our goal-seeking analysis.

Fig. 2-8-6.
A child window "Cfin1" shows the result of the goal-seek analysis based on the "Finance Model".

Fig. 2-9-1.
A child window titled "Reasoning1" has four items as the submenu.

Fig. 2-9-2.
In the child window "Report.txt", "Report.txt" file is read in "Evaluation&Report" edit box, "Cfin1.dat" file is read in the "Cfin1.data" edit box, and "Cfin1.pro" file is incorporated into "Analysis.pro" program file in the "Analysis.pro" edit box.

Fig. 2-9-3.
A new window titled "Analysis.pro" supervised by Prolog compiler, "Strawberry Prolog". In this Prolog window the "Analysis.pro" program is shown on the left and the messages (i.e. the results for the reasoning) sent back from ES to the output window on the right.

Fig. 2-10-1.
A dialog box "Conditions of Drawing Graph" establishes various setups for drawing the line chart.

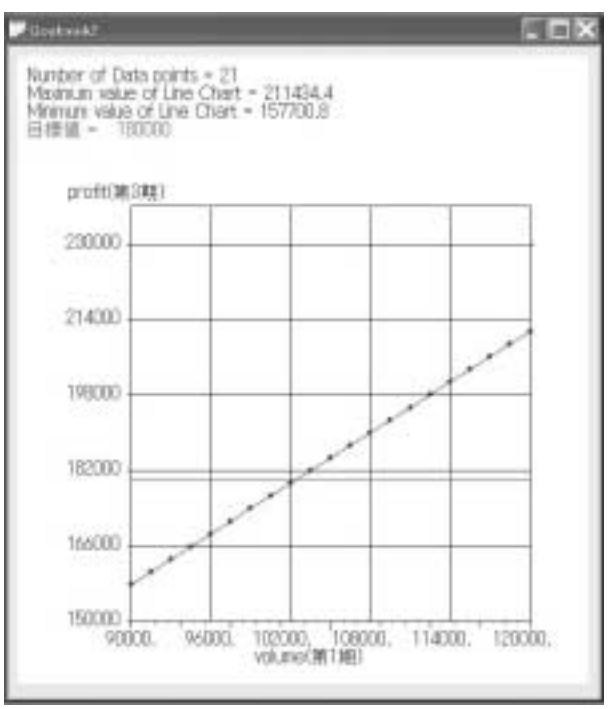

Fig. 2-10-2.
A graph plotting the goal-seek simulation results, with the abscissa specifying the output variable and the ordinate specifying the exogenous input variable. The desired value of the goal we set up is also shown with a red line.

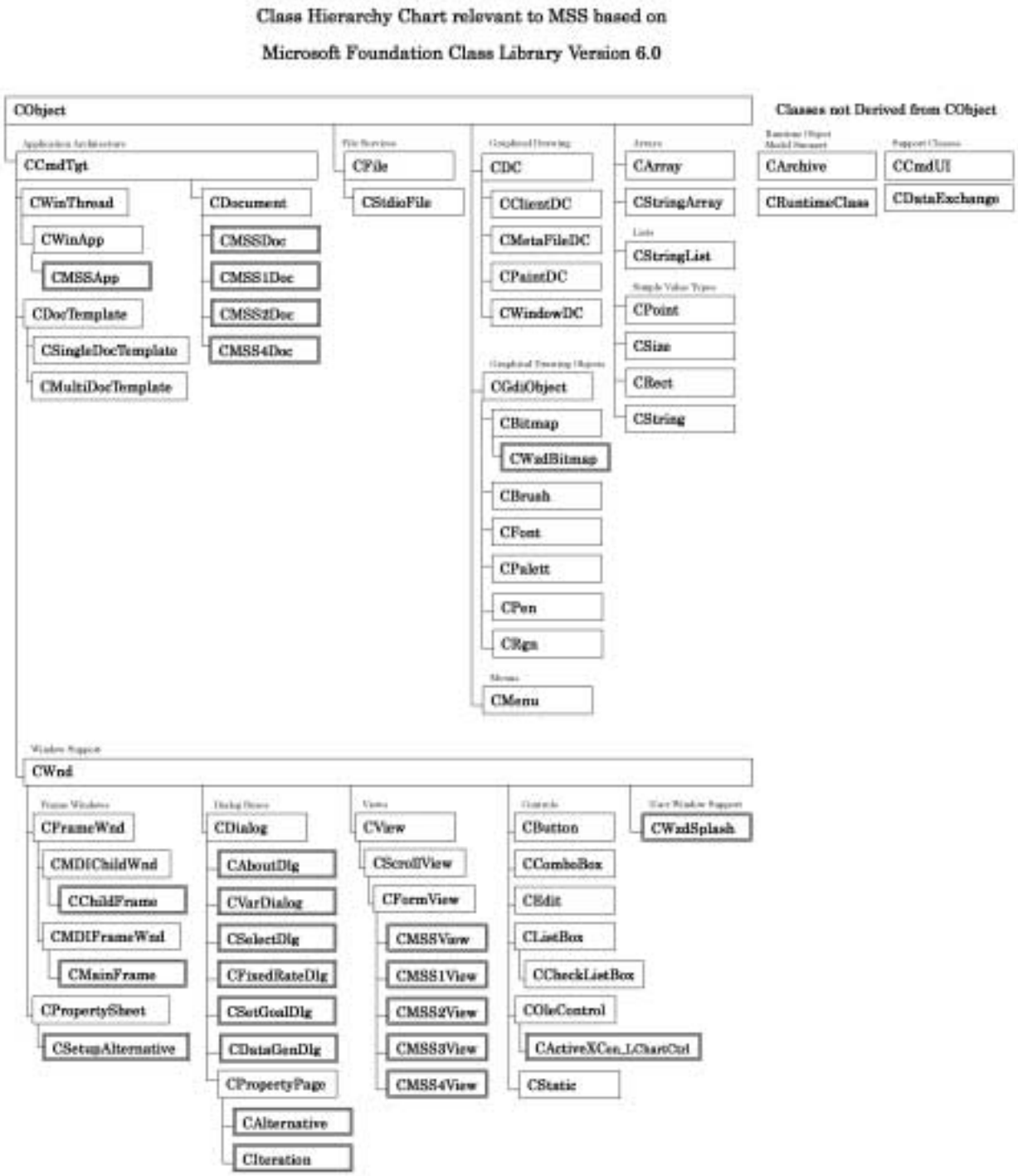

Fig. 4.
Class hierarchy chart relevant to MSS based on Microsoft Foundation Class Library Version 6.0.
Our "user-defined classes" are displayed as double squares.

[Abstract]

Development of a Management Support System On the Windows Platform (I) :
Class structure of MFC and creation of user-defined classes

Hiroshi NOTO

We present a management support system(MSS) developed in our 'Seminar I' and 'Seminar II' in the Management and Information Department of Hokusei Gakuen University. The MSS written in Visual C++ basically exploits MFC class library. MFC defines the structure of our application and manipulates routine works on the application's behalf. We review the basic concepts of MFC and give a run-down of classes grouped into underlying categories. After looking over the design and performance of our MSS, we elaborate relevant user-defined classes the MSS creates which are derived from their respective MFC base classes, each functionally grouped into categories according to the class hierarchy chart of the MFC class library. We find that the MSS is realized basically exploiting the class structure of MFC of which the functionalities are inherited to our user-defined classes and which makes the Windows programming much more efficient, transparent and rewarding.

正　誤　表

北星論集経済学部第42巻第２号

頁・行目	誤	正
３頁　下から　13行目	GDI objects	GDI object
６頁　下から　８行目	your input	our input
８頁　下から　１行目	Likewiseから始まる段落は直前の段落に含めて続ける。	
９頁　下から　15行目	a exogenous	an exogenous