Ottobre 1997

CORSO DI PROGRAMMAZIONE ORIENTATA AGLI OGGETTI IN VISUAL C++
e
UTILIZZO DI MICROSOFT FOUNDATION CLASS

Ing. Stefano Riccio
stefaric@tiscalinet.it

BIBLIOGRAFIA

🕮 Per quanto riguarda la programmazione in 'C++' utilizzando MFC:

- **Manuali Visual C++**
- **Microsoft Visual C++, Programmare in MFC e Win 32**
  ed. Mondadori Informatica
- **Il manuale Visual C++**
  David J. Kruglinski ed. McGraw Hill

🕮 Altri testi utili:

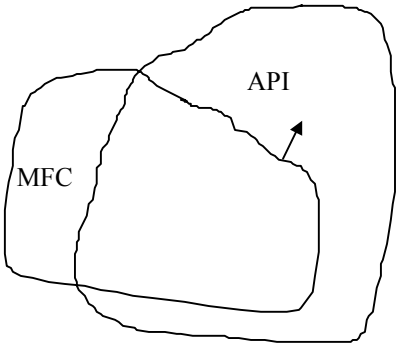
- J. Conger, Microsoft Foundation Class Primer, Wait Group Press, 1993
- B. Struostrup, Il linguaggio C++, Addison Wesley, 1993
- S.Holzner, Programmazione in Visual C++, Jackson libri, 1993
- C Simonelli - C. Munisso, Dal C aWindows passando per il C++, 1995

Diversi livelli di utilizzo di Visual C++

-utilizzo programmazione ad oggetti usando MFC (documento-vista) e strumenti di ausilio al programmatore (Application Wizard, ClassWizard)
-utilizzo programmazione ad oggetti usando MFC con documento-vista
-utilizzo programmazione ad oggetti usando MFC senza documento-vista
-utilizzo programmazione ad oggetti (C++) senza usare Microsoft Foundation class (MFC)
-utilizzo SDK (programmazione in C)

Il corso sara' rivolto all'utilizzo del C++ e di MFC e degli strumenti dell'Application Framework. Vedremo qualche esempio di utilizzo della programmazione ad oggetti senza l'architettura Documento-Vista (se non si e' interessati ad avere la serializzazione, il supporto di stampa e le barre di controllo si puo' risparmiare 30Kb di codice e non utilizzare documento vista).

Relazione tra API e MFC



MFC purtroppo non copre tutte le api di windows, quindi il programmatore esperto spesso ha bisogno di funzioni non contemplate nelle MFC e deve utilizzare le API di windows.

A questo scopo ogni classe ha un dato membro contenente l'HANDLE della finestra (m_hWnd di CWnd) ed e' sempre possibile utilizzare l'operatore di scope "::" per accedere alle funzioni globali della API che normalmente si utilizzano con SDK.

La gerarchia delle classi (VEDERE GRAFICI SUCCESSIVI)

La gerarchia di classi di MFC e suddivisa in 5 gruppi:

-Application architecture classes
-Visual Object Classes
-General Purpose Classes
-Database Classes
-OLE 2 Classes

Queste 5 famiglie forniscono le seguenti funzioni:
-classi di collezioni per liste, array, mappe
-una classe stringa
-classi tempo, spazio, data
-classi di accesso ai file (di testo o binari)
-supporto OLE 2
-supporto ODBC
-gestione eccezioni
-inclusione controlli VBX in applicazioni C++
inoltre per sistemi a 32 bit gestisce i thread (processi figli concorrenti)
ecc...

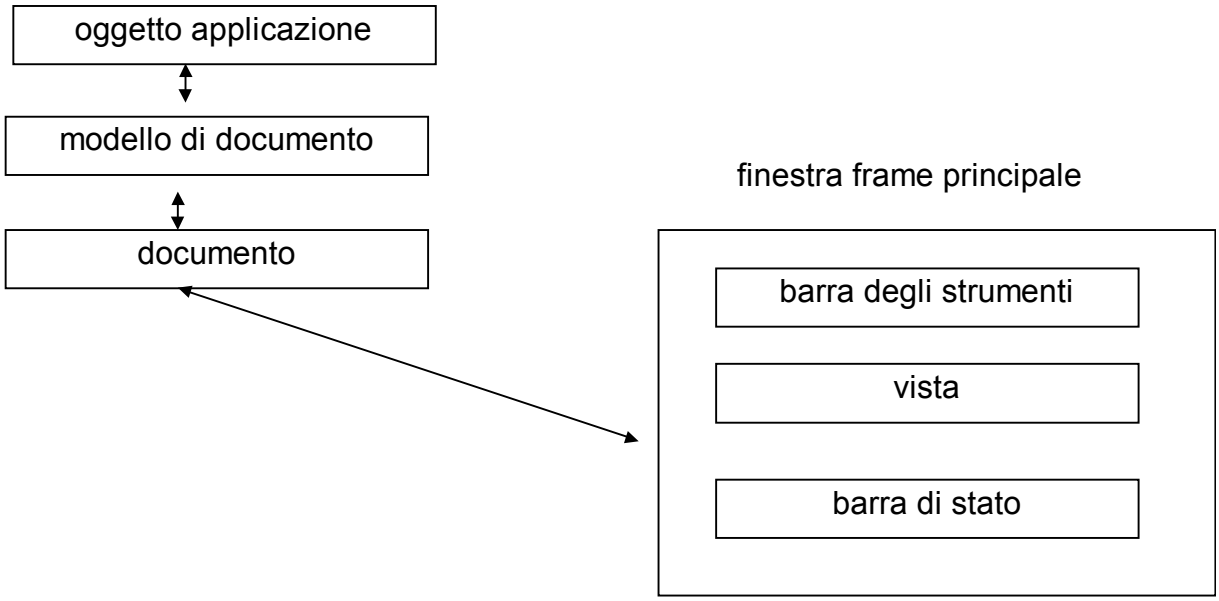
Pregi e difetti delle MFC

Utilizzando MFC si raggiunge la massima astrazione possibile dall'hardware e dal sistema operativo, basti pensare al fatto che nel passaggio da Windows 3.1 a windows 95 non bisogna riscrivere il codice sorgente, basta ricompilare e le classi saranno a 32 bit (ATTENZIONE: sempre che il programmatore abbia utilizzato esclusivamente le classi di MFC (cosa decisamente improbabile).
Utilizzando MFC si diventa praticamente dipendenti da MFC, infatti se si volesse continuare a programmare in C++ in un altro ambiente (es. Borland) tutto cio' che si e' imparato non serve piu' a nulla perche' cambia la gerarchia di classi (pur essendo lo stesso C++).
Comunque MFC a differenza di OWL di Borland sembra stia diventando lo standard di fatto per le gerarchie di classi in ambienti a finestre.

Architettura delle MICROSOFT FOUNDATION CLASS

Le classi di MFC formano un framework di applicazione, esso definisce l'ossatura di una applicazione e fornisce delle realizzazioni standard di interfaccia utente che possono essere poste sull'ossatura. Il compito del programmatore e' scrivere la parte rimanente dell'ossatura, quelle cose che sono specifiche della propria applicazione.
Gli oggetti chiave in una applicazione in esecuzione sono:
-il/i documento/i (derivato da CDocument)
-la/le vista/e (derivata da Cview)

-la finestra frame (derivata da CFrameWnd per SDI e da CMDIFrameWnd per le MDI per la finestra madre; derivata da CMDIChildWnd per le finestre figlie in una MDI)
-il/i modello/i di documento
il suo compito e' di orchestrare documento, vista e finestre frame (si utilizza un oggetto CSingleDocTemplate per SDI e CMultiDocTemplate per MDI)
-oggetto applicazione (derivato da CWinApp)
esso e' unico e solo, contiene tutti gli altri oggetti e rappresenta l'applicazione stessa



In questa figura si vede la relazione fra i vari oggetti in una applicazione in esecuzione.

L'architettura documento-Vista

Qualsiasi programmatore, quando scrive un programma, memorizza i dati di cui ha bisogno in determinate strutture dati. Poi non fa altro che rappresentare questi dati sul video in modo comprensibile per l'utente finale. Spesso la rappresentazione a video dei dati e' diversa dalla reale struttura di memorizzazione dei dati stessi (ad esempio un array di valori puo' essere rappresentato a video da un diagramma a torta).
Su questo principio si basa l'architettura documento-vista. Io ho il sorgente suddiviso in 2 fondamentali parti:
-il documento contenente la struttura dati che utilizzo
-la vista che gestisce la rappresentazione di tali dati a video (ma non solo ...)

Nella programmazione ad oggetti questi 2 parti di software diventano due classi. L'oggetto documento avra' solo ed esclusivamente funzioni relative alla gestione dei dati (routine di ordinamento, di ricerca, routine di caricamento, salvataggio ecc...). L'oggetto vista avra' solo ed esclusivamente funzioni di interfaccia con l'utente; avra' inoltre un puntatore al documento in modo che possa in qualsiasi momento accedere ai suoi dati.

E' chiaro che io posso avere piu' viste per ogni documento.

Come il framework chiama il vostro codice

Quando l'applicazione e' in esecuzione la maggior parte dei flussi di controllo risiede nel codice del framework. Il framework gestisce il message loop (il programmatore non lo deve piu' scrivere). Eventi che il framework gestisce da solo non dipendono piu' dal vostro codice. Mentre gestisce questi lavori il framework utilizza gestori di messaggio e funzioni virtuali di C++ per darvi la possibilita' di rispondere a determinati eventi (click del mouse, pressione di un bottone, scelta di una voce di menu' ecc...). Ad esempio quando si sceglie una voce di menu il framework instrada il messaggio attraverso tutti gli oggetti del programma, nel punto in cui voi avete derivato una funzione membro per la gestione di quel messaggio, arrivera' il vostro messaggio e potra' essere eseguito il vostro codice personalizzato.

CWinApp : la classe applicazione

Incapsula l'inizializzazione, l'esecuzione e la fine di una applicazione windows. Dentro questa classe e' contenuta la funzione WinMain (ogni applicazione windows deve avere questa funzione, ma voi non la dovete scrivere, e' gia contenuta in CWinApp). La funzione WinMain chiama nell'ordine:
-InitInstance, funzione virtuale scavalcabile che inizializza l'istanza corrente dell'applicazione
-Run ,funzione virtuale che contiene il message loop, se volete personalizzare il message loop dovete scavalcare questa
-ExitInstance ,per fare le operazioni di pulizia
-OnIdle

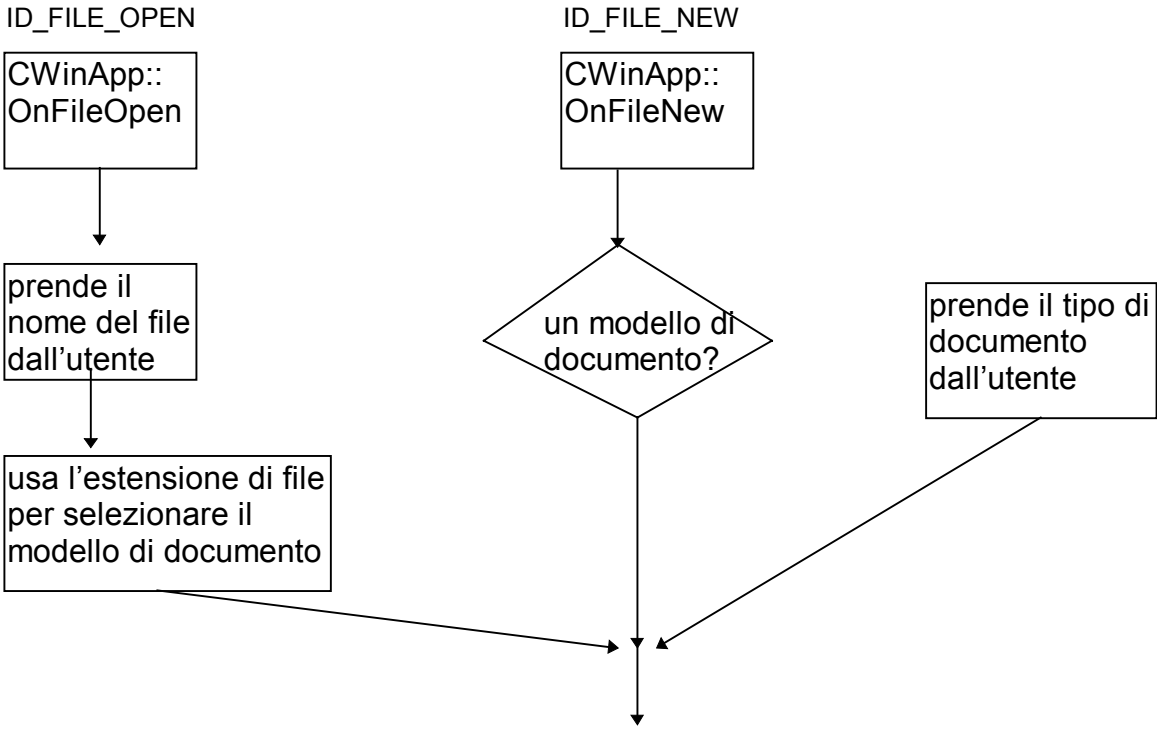
La CWinApp contiene anche le funzionalita' molto interessanti:
-aggancio al file manager
-drag and drop con il file manager
-gestione del file .ini

I modelli di documento

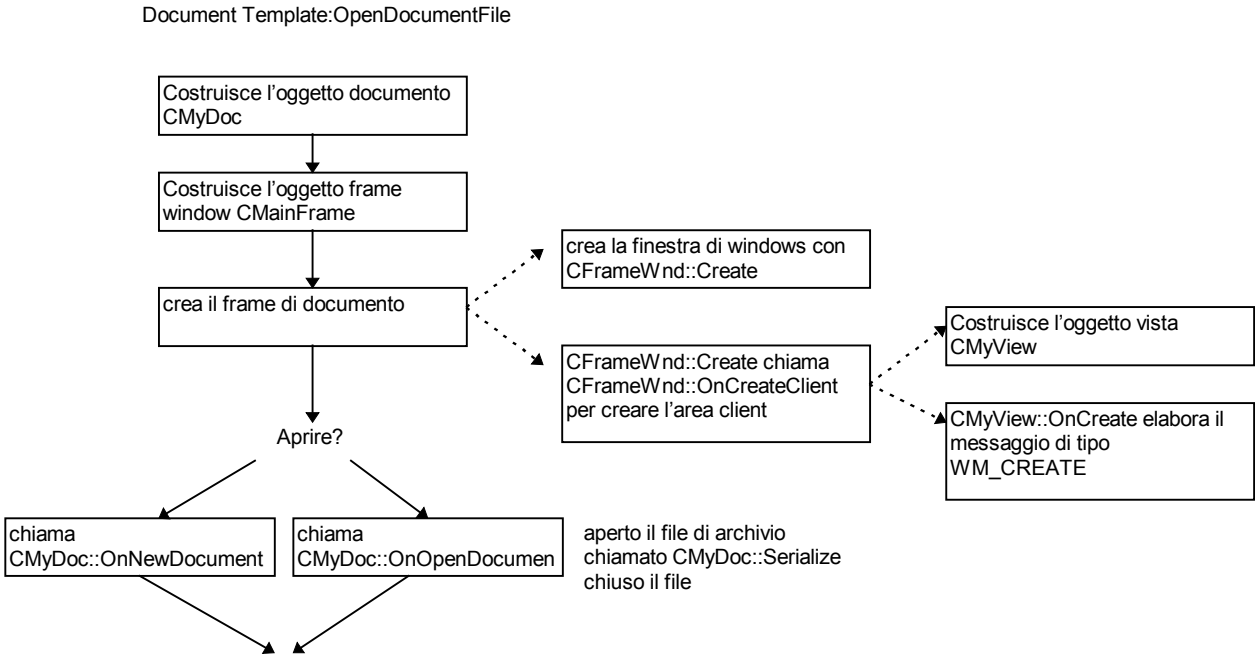
Per gestire il complicato processo di creazione dei documenti con le loro viste e finestre frame associate il framework utilizza due classi di modello di documento:
-CSingleDocTemplate per SDI (puo' gestire un documento di un tipo per volta)
-CMultiDocTemplate per MDI (tiene una lista di documenti tutti dello stesso tipo)
Se una applicazione gestisce piu' tipi di documenti, ogni volta che si scegli NEW appare una dialogbox che chiede il tipo di documento che si vuole utilizzare.
L'oggetto applicazione mantiene una lista dei modelli di documento (i tipi diversi), questa lista si crea aggiungendo voci tramite AddDocTemplate.

Processo di creazione del documento e della vista

oggetto applicazione ----------> crea ----------> modello di documento
modello di documento----------> crea ----------> documento
finestra frame
finestra frame ----------> crea ----------> vista



Sequenza di creazione di un documento



Sequenza di creazione di una finestra frame

Finestre suddivise e viste multiple

Abbiamo detto che e' possibile gestire piu' documenti (attraverso le MDI), piu' viste per ogni documento e piu' modelli di documento. Come si puo' intuire le combinazioni possibili

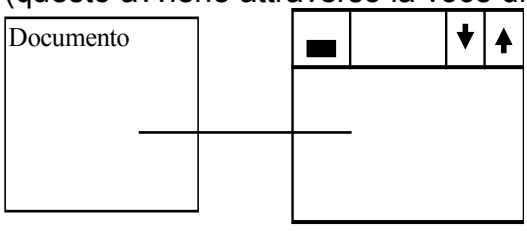
sono tante, ed esse aumentano se consideriamo la possibilita' (offerta da MFC) di utilizzare le finestre suddivise.
Le finestre suddivise sono finestre che possono contenere piu' viste diverse dello stesso documento all'interno di esse. Cio' e' possibile mediante dei separatori all'interno della finestra (un esempio di finestra suddivisa e' File Manager di Windows).

Cerchiamo allora di fare un po' di ordine elencando le possibili combinazioni:
1) SDI (un solo modello di documento)
-un solo documento, finestra suddivisa ma una sola vista
-un solo documento, finestra suddivisa con piu' viste (tipi di viste diverse)
2) MDI (un solo documento e una sola vista
-un solo modello di documento)
-un solo documento, una sola vista
-un solo documento con piu' viste (utilizzando la voce di menu' NUOVA FINESTRA)
-un solo documento, finestra suddivisa ma una sola vista (ma anche piu' finestre con la suddetta voce di menu)
-un solo documento, finestra suddivisa con piu' viste (ma anche piu' finestre con la suddetta voce di menu)
3) MDI (piu' modelli di documento)
Qui posso avere contemporaneamente aperti diverse combinazioni del caso 2. Ognuna di quelle combinazioni rappresenta un modello di documento. Per fare questo basta creare ad esempio altre voci di menu NUOVA FINESTRA, in risposta alle quali si "carica" un nuovo modello di documento, e quindi una nuova vista, sempre sullo stesso documento.

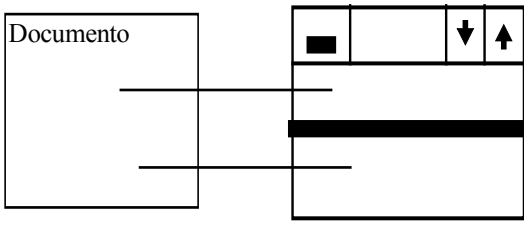
Da queste combinazioni manca l'idea di una vista aperta su piu' documenti !!!.

OSSERVAZIONI:
-E' facile creare nuovi tipi di modelli di documento (basta creare le classi documento). Per far sapere al framework (o meglio alla finestra frame principale) che e' stato creato un nuovo modello di documento bisogna aggiungere una chiamata a AddDocTemplate nell'override InitInstance della classe della vostra applicazione.
-E' possibile associare piu' viste allo stesso documento. Prima di tutto occorre creare le varie classi vista, dopodiche' bisogna informare la classe documento della presenza di queste viste ad esso associate. A questo scopo ogni classe documento contiene dentro di se una lista di puntatori alle sue viste, il programmatore deve aggiornare questa lista a seconda di quante viste vuole associare al documento.
-In pratica delle varie combinazioni viste, nel caso di piu' viste, sono supportate solamente 3 possibilita' con MFC -corrispondenti a 3 modalita' comuni di interfaccia utente-

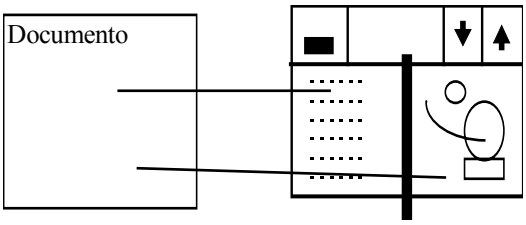
1) In una MDI posso aprire piu' finestre figlie, ognuna delle quali ha una propria vista (tutte dello stesso tipo), ma tutte agiscono sullo stesso documento. Come se in Word apro due volte la stessa lettera e visualizzo nelle 2 finestre parti diverse dello stesso documento. (questo avviene attraverso la voce di menu NEW WINDOW)



2) Uso una finestra suddivisa, dove ho due viste della stessa classe, ambedue agiscono sullo stesso documento.



3) Uso una finestra suddivisa, dove ho 2 viste di due classi diverse che agiscono sullo stesso documento.



Collegamenti tra gli oggetti

La seguente tabella mostra come e' possibile, per il proogrammatore, accedere a tutti gli oggetti presenti in una applicazione:

DA \ per OTTENERE	**vista**	**documento**	**finestra frame**	**Modello di doumento**
vista	**********	GetDocument	GetParentFrame	*attraverso il documento*
documento	GetFirstViewPosition GetNextView	**********	*attraverso la vista*	GetDocTemplate
finestra frame di documento	GetActiveView	GetActiveDocument	**********	*attraverso il documento*
finestra frame MDI	*attraverso la finestra frame*	*attraverso la finestra frame*	MDIGetActive	*attraverso la finestra frame*

Qualsiasi oggetto puo' accedere alla APPLICAZIONE con **AfxGetApp**
l'APPLICAZIONE raggiunge ogni oggetto attraverso il modello di documento che viene creato nell'oggetto applicazione stessa
il modello di documento contiene dentro di se tutti i puntatori agli oggetti esistenti

I messaggi in MFC

In MFC una funzione "gestore" dedicata elabora ogni singolo messaggio (in SDK era un CASE della istruzione SWITCH). Le funzioni gestore di messaggio sono delle funzioni membro di una classe. ClassWizard pensa a scrivere i prototipi e le intestazioni delle funzioni gestore di messaggio, a noi il compito di scrivere il codice all'interno delle funzioni.
I messaggi gestiti sono di 3 tipi:
-messaggi di windows (WM_xxxx tranne WM_COMMAND)
-notifiche di controllo (WM_COMMAND da controlli e finestre figlie)
-messaggi di comando (WM_COMMAND da oggetti di interfaccia utenti)

Ogni classe di framework che puo' ricevere messaggi possiede la propria "mappa dei messaggi". Queste mappe di messaggi servono al framework per collegare i messaggi alle loro funzioni di gestore.
Di solito tutti i messaggi arrivano alla finestra frame principale, ma i messaggi di comando vengono INSTRADATI ad altri oggetti. Il framework si occupa di instradare i messaggi

attraverso una sequenza standard di oggetti cosiddetti di “obiettivo-comando”. Ogni oggetto di obiettivo-comando controlla la propria mappa dei messaggi per vedere se possiede dentro di se una funzione gestore per quello specifico messaggio. Se l’oggetto obiettivo-comando non e’ in grado di trattare quel messaggio lo instrada verso altri oggetti obiettivo-comando secondo una sequenza ben stabilita (si puo’ trovare una tabella della suddetta sequenza sui manuali o sulla Enciclopedia OnLine).

Per eseguire questo instradamento di comandi, ogni oggetto obiettivo-comando chiama la funzione OnCmdMsg dell’oggetto obiettivo-comando successivo nella sequenza. Il programmatore puo’ scavalcare la funzione OnCmdMsg per fare un instradamento personalizzato (aspetto molto avanzato di programmazione).

Utilizzando AppWizard e ClassWizard per associare i messaggi ai gestori il programmatore non si deve preoccupare di creare le funzioni per fare l’instradamento, perche’ fa tutto il framework. Basta osservare il file .CPP contenente la mappa dei messaggi delimitata da due macro predefinite:
BEGIN_MESSAGE_MAP(...)
END_MESSAGE_MAP()

Attenzione: se una classe non possiede una mappa dei messaggi, cio’ non vuole dire che essa non possa gestire dei messaggi. Infatti le mappe dei messaggi si ereditano, quindi ogni classe “vede” tutti i messaggi delle classi progenitrici nella gerarchia.

COME SI SCRIVONO I GESTORI DEI MESSAGGI
Una volta creata una voce all’interno della mappa dei messaggi (che associa a un determinato messaggio una specifica funzione gestore) bisogna creare una funzione gestore. Questa e’ una funzione membro della classe, come ad esempio:

```
afx_msg void OnPaint();
```

afx_msg e’ una parola chiave che indica che quella e’ una funzione gestore. Potete pensare a afx_msg come una macro del tipo
#define afx_msg virtual (IN REALTA’ NON E’ COSI)
quindi serve solo al programmatore per distinguere le funzioni gestore dalle altre funzioni membro.

La serializzazione

I documenti contengono e gestiscono i dati della vostra applicazione. Quindi dovrebbero, anche occuparsi della loro memorizzazione sul disco. Per facilitare questo compito al programmatore, i progettisti della Microsoft hanno creato il meccanismo della Serializzazione.
La classe CDocument ha una funzione membro chiamata Serialize che viene automaticamente chiamata quando l’utente sceglie le voci di menu APRI,SALVA,SALVA CON NOME. Essa si occupa di caricare o di riversare i dati sul disco; oltre alle solite aperture e chiusure del file (ovviamente).
Il programmatore puo’ scavalcare questa funzione per personalizzare il caricamento e il salvataggio.
Ma la grossa forza di questo meccanismo e’ un’altra.
Se i dati contenuti nel documento sono tutti contenuti in oggetti derivati da Cobject e possibile demandare le operazioni di I/O a queste strutture dati.

Il concetto di fondo e' "ognuno carica e salva per se".
Ad esempio: se il documento contiene un array, la Cdocument chiamera' la serialize di quel array. A sua volta la Serialize di quell'array chiamera' la serialize di ogni elemento dell'array. Finalmente ogni elemento dell'array salvera' su disco i propri dati.

La serializzazione ha molti limiti, ma risulta comunque un utile strumento per gestire facilmente l'I/O. Quindi a mio avviso conviene usarla quando non sia ha bisogno di particolari prestazioni.

Servizi Cobject

La classe base Cobject fornisce agli oggetti delle sue classi derivate i seguenti servizi:
-diagnostica di oggetto
-informazioni di classe runtime
-persistenza di oggetto

diagnostica di oggetto

E' un meccanismo utile al programmatore per facilitare le operazioni di debug. Ci sono delle macro e delle funzioni che permettono di visualizzare sul video secondario (WinDebug) dei dump di memoria per capire dove ci sono degli errori.
Ci sono statistiche di allocazione di memoria (utilizzando un operatore new di debug particolare).
TRACE agisce come una printf sul video secondario, il codice relativo a questa funzione non viene chiaramente generato nelle compilazioni di release.
Si puo' utilizzare un riversamento degli oggetti "stile flusso" utilizzando afxDump.
ASSERT valuta una condizione specifica, se la condizione e' falsa genera un messaggio di errore.
VERIFY ha lo stesso funzionamento di ASSERT solo che viene mantenuta nel codice di release.
AssertValid e' una funzione di Cobject che serve per verificare la validita' interna dei dati.

informazioni di classe runtime

E' un meccanismo che permette al programmatore di controllare il tipo di una variabile, di un oggetto in fase di esecuzione. Per poter utilizzare queste potenzialita' occorre:
-derivare la propria classe da Cobject
-utilizzare le seguenti macro:
DECLARE_DYNAMIC, DECLARE _DYNCREATE, nella dichiarazione della classe (file .H)
IMPLEMENT_DYNAMIC,IMPLEMENT_DYNCREATE nella realizzazione della lasse (file .CPP)

persistenza di oggetto

La persistenza degli oggetti permette di salvare una rete complessa di oggetti in una forma binaria permanente su disco. Questo avviene con il meccanismo della serializzazione.
Per creare una classe serializzabile occorre:
-derivare la propria classe da Cobject
-utilizzare le seguenti macro:

DECLARE _SERIAL nella dichiarazione della classe (file .H)
IMPLEMENT_SERIAL nella realizzazione della classe (file .CPP)

Le classi di collezione

MFC contiene un certo numero di liste, array e mappe pronti all'uso a cui ci si riferisce con il termine "classi di collezione".
MFC fornisce due tipi di classe di collezioni:
-modelli di collezione (SOLO VISUAL C++ 2.0 e oltre)
-collezioni di tipo non modello (vediamo solo queste)

Vediamo come e' possibile accedere a tutti i membri di una collezione:

PER ITERARE UN ARRAY

```
CPtrArray myarray;
for(i=0;i<myarray.GetSize();i++)
{
 Cperson *pperson=(CPerson *)myarray.GetAt(i);
 ...
}
```

PER ITERARE UNA LISTA

```
CPtrList mylist;
POSITION pos=mylist.GetHeadPosition();
while(pos!=NULL)
{
 Cperson *pperson=(CPerson *)mylist.GetNext(i);
 ...
}
```

PER ITERARE UNA MAPPA

```
CMapStringToOb mymap;
POSITION pos=mymap.GetStartPosition();
while(pos!=NULL)
{
 Cobject *pObject;
 Cperson *pperson;
 Cstring string;
 mymap.GetNextAssoc(pos,string,pObject);
 pperson=(* Cperson)pObject;
 ...
}
```

Vediamo come e' possibile eliminare tutti i membri di una collezione (cosa che di solito si fa nel distruttore dell'elemento) :

PER ELIMINARE UN ARRAY

```
CPtrArray myarray;
i=0;
while(i<myarray.GetSize())
{
 delete (CPerson *)myarray.GetAt(i++);
}
myarray.RemoveAll();
```

PER ELIMINARE UNA LISTA

```
CPtrList mylist;
POSITION pos=mylist.GetHeadPosition();
while(pos!=NULL)
{
 delete (CPerson *)mylist.GetNext(i);
}
mylist.RemoveAll();
```

PER ELIMINARE UNA MAPPA

```
CMapStringToOb mymap;
POSITION pos=mymap.GetStartPosition();
while(pos!=NULL)
{
 Cobject *pObject;
 Cperson *pperson;
 Cstring string;
 mymap.GetNextAssoc(pos,string,pObject);
 pperson=(* Cperson)pObject;
 delete pperson;
}
mymap.RemoveAll();
```

OSSERVAZIONE : e' immediato costruire degli oggetti STACK e CODA partendo da queste strutture

Oggetti di interfaccia utente

Gli oggetti di interfaccia utente sono:
-voci di menu
-pulsanti della barra degli strumenti
-tasti di accelerazione

Essi sono oggetti che generano dei comandi, ai quali sono associati degli identificatori. Ogni volta che l'utente compie un'azione sull'interfaccia grafica, vengono svolti i seguenti compiti all'interno del framework:
-viene selezionato un oggetto dell'interfaccia utente (UI)
-l'oggetto UI manda il messaggio WM_COMMAND
-il messaggio viene instradato e ognuno lo cerca nella propria mappa dei messaggi
-chi lo trova richiama la funzione associata
-viene eseguito la funzione scritta da voi in risposta a quel messaggio

Il framework si occupa anche di aggiornare gli oggetti UI, cioe' abilitarli o disabilitarli (renderli grigi) a seconda dei casi. Prima di disegnare il menu a discesa, oppure nei momenti di inattivita' nel caso dei pulsanti della barra degli strumenti, il framework spedisce un comando di aggiornamento. Se voi riuscite a mappare questo messaggio di aggiornamento con una vostra funzione allora potete compiere le operazioni necessarie ad intervenire sull'oggetto UI.
Facciamo un esempio:
quando l'utente sceglie di aprire il menu' a tendina FILE, il framework instrada prima tutti i messaggi di aggiornamento di tutte le voci di menu della tendina che deve disegnare. Successivamente disegna il menu' a tendina. Allora il programmatore puo' derivare (con classWizard) un messaggio di tipo ON__UPDATE_COMMAND_UI; creare una sua funzione nella quale puo' facilmente disabilitare le voci di menu, poiche' puo' sfruttare i metodi offerti dall'oggetto UI. Se il framework non trova una funzione associata all'aggiornamento ABILITA la voce per default.
Quando si deriva un messaggio di aggiornamento ClassWizard scrive gia' una funzione che ci fornisce un puntatore all'oggetto UI (CCmdUI* pCmdUI), con il metodo Enable si puo' abilitare o no la voce di menu.

Barre di controllo

La classe Base che gestisce le barre di controllo e' CControlBar.
Da questa derivano 3 classi:
-CToolBar per le barre degli strumenti
-CStatusBar per le barre di stato
-CDialogBar per le barre di dialogo

Barre degli strumenti
A ogni ToolBar e' associato un bitmap contenente tutte le icone dei pulsanti.
E' un unico bitmap con tante icone di 15 pixel di altezza e 16 di base. Il framework si occupa di fare il bottone e di cambiare aspetto quando il bottone e' premuto o disabilitato. Questi pulsanti sono collegati agli identificatori tramite un array nel quale la posizione dell'identificatore dell' array corrisponde alla posizione dell'immagine nel bitmap. Se in questo array si inserisce ID_SEPARATOR si ottengono pulsanti separati.
Per default la barra degli strumenti e' attaccata in cima alla finestra, ma puo' essere agganciata a qualsiasi lato o rimanere fluttuante.
Per fare questo bisogna abilitare la finestra frame all'aggancio con CFrameWnd::EnableDocking,
bisogna abilitare all'aggancio la barra con CControlBar::EnableDocking,
se non si indica nessun lato la barra sara' fluttuante.
Per effettuare effettivamente la barra si deve chiamare CFrameWnd:: DockControlBar, ma questo in genere viene fatto dal framework quando l'utente lascia cadere la barra vicino ai lati della finestra cornice. Con CFrameWnd::FloatControlBar si puo' staccare dai lati della cornice.
S possono salvare le informazioni di come sono posizionate le barre nel .INI con SaveBarState e LoadBarState.
Essa puo' anche gestire dei suggerimenti sui pulsanti.
Per fare cio' bisogna aggiungere lo stile CBRS_TOOLTIPS alla funzione Create di CtoolBar; successivamente bisogna aggiungere (con appstudio) ad ogni identificatore la stringa di aiuto nella casella prompt preceduta da '\n'.

Barre di stato
In questo caso non e' presente il BITMAP. Pero' e' presente un array di identificatori associati ai campi della barra.
ID_INDICATOR_CAPS, ID_INDICATOR_NUM, ID_INDICATOR_SCRL sono identificatori predefiniti che si possono utilizzare.
E' possibile accedere ai vari campi della barra di stato mediante la funzione CStatusBar::SetPaneText

Barre di dialogo
Sono lo strumento piu' potente in quanto si possono inserire dentro di esse tutti i controlli di una dialog box direttamente da una risorsa di dialogo. In pratica e' molto simile ad una dialog box non modale solo che ha una MINI cornice (CMiniFrameWnd).

OSSERVAZIONE: anche per i pulsanti delle barre ci sono i messaggi di aggiornamento UI, in questo caso si utilizza la funzione CCmdUI::SetCheck per mettere nello stato di premuto o no il relativo bottone.

Gestione delle ECCEZIONI

In C++ e' stato creato un meccanismo particolare per gestire tutti gli eventi asincroni rispetto al normale svolgimento di un programma. In qualsiasi punto del programma ci puo' essere una funzione che fa operazioni potenzialmente fonte di errore. Ad esempio errori sui file, errori di memoria esaurita, errori di risorse non disponibili ecc...

Nelle prime versioni di C++ erano presenti delle macro per gestire le eccezioni: TRY,CATCH,END_CATCH,THROW. Ora queste macro sono ancora supportate per motivi di compatibilita' con le vecchie applicazioni, ma in realta' sono state sostituite da parole chiave del linguaggio: try,catch,throw.

COME CATTURARE ED ELIMINARE ECCEZIONI:

Ogni volta che si deve eseguire una operazione che puo' dare qualche problema bisogna racchiudere la stessa operazione nel blocco try, nel seguente modo:

```
try
{
.....
}
```

Questo prepara il compilatore ad indirizzare eventuali errori a delle classi predefinite, create apposta per gestire tali eccezioni.
Queste classi sono:

- CMemoryException
- CFileException
- CArchiveException
- CNotSupportedException
- CResourceException
- CDBException
- COleException
- COleDispatchException
- CUserException

A questo punto il programmatore puo' catturare queste eccezioni usando la parola chiave catch, questo lo si puo' fare localmente di seguito alla parola chiave try, ossia scrivendo questo blocco:

```
catch(CException *e)
{
.....
e->Delete();
}
```

Nel caso non si sia verificata alcuna eccezione, il blocco catch viene ignorato. Nel caso si sia verificata una eccezione viene richiamato il blocco catch e sara' di seguito eseguito il codice interno al blocco catch stesso. Il programmatore ottiene dal framework un puntatore alla classe che gestisce l'errore ("e"), questo oggetto puo' dare delle informazioni utili a capire cosa e' successo.
E' obbligatorio cancellare "e" alla fine del blocco catch, perche' si deve deallocare la sua memoria.
E' possibile scrivere piu' blocchi catch dopo un blocco try purche' ogni blocco catch gestisca un particolare tipo di errore. Il seguente codice cattura le eccezioni dei file e della memoria, oltre agli altri tipi generici.

```
try
{
.....
}
catch(CFileException *e)
{
.....//gestisce gli errori dei files
e->Delete();
}

catch(CMemoryException *e)
{
.....//gestisce gli errori di memoria
e->Delete();
}
catch(CException *e)
{
.....//gestisce gli alti errori
e->Delete();
}
```

OSSERVAZIONE : e' importante deallocare tutto cio' che e' stato allocato nello heap quando si entra nel blocco catch.

COME RICHIAMARE UNA ECCEZIONE:

E' anche possibile richiamare una eccezione da una nostra funzione. Per fare cio' basta fare un test e poi richiamare una delle funzioni apposite di MFC (ad es. AfxThrowMemoryException() per le eccezioni di memoria). Ad esempio, la seguente procedura controlla se si e' riusciti ad allocare la memoria, se c'e' stato un errore lancia una eccezione.

```
{
 char *p=(char *)malloc(1000);
 if(p=NULL)
 {
 AfxThrowMemoryException();
 }
```

LA STAMPA con MFC

COME AVVIENE LA STAMPA DI DEFAULT DEL FRAMEWORK

La vista possiede una funzione OnDraw che riceve un puntatore a CDC. Quando il framework riceve un messaggio WM_PAINT (sintomo dell'esigenza di disegnare sul video) il framework chiama OnDraw e gli passa un puntatore al display context associato alla vista, di conseguenza l'output va sul video. Quando invece si deve stampare il framework passa alla OnDraw il puntatore a un device context gia' mappato con la stampante. Ecco perche' quando si scrivono applicazioni con MFC si ottiene "gratis" tutta la stampa.
Tutto questo va bene finche' si vuole una stampa esattamente identica alla vista (anche nelle dimensioni). Se si vuole stampare con dimensioni diverse oppure se si vuole stampare su piu' pagine allora bisogna modificare il comportamento di default del framework intervenendo direttamente sul codice.

IL PROCESSO DI STAMPA DEL FRAMEWORK

La classe vista richiama in sequenza tutta una serie di funzioni membro per gestire il complesso processo di stampa. Il programmatore puo' intervenire in questo processo semplicemente andando a ridefinire quelle funzioni che servono a personalizzare il processo stesso.
Ecco la sequenza delle funzioni richiamate:

- CMyView::OnPreparePrinting
  decide la suddivisione logica del documento in pagine
  imposta la lunghezza del documento (se e' nota) attraverso la SetMaxPage
  richiama DoPreparePrinting per visualizzare la finestra di dialogo e creare il device context (appare a video la finestra classica per la stampa)
- CMyView::OnBeginPrinting
  Alloca le risorge GDI per la stampa
- CDC::StartDoc
- CMyView::On PrepareDC (richiamata per ogni pagina)
  Cambia l'origine del viewport e il modo di mapping delle coordinate logiche
  se la lunghezza del documento non e' stata specificata controlla la fine del documento

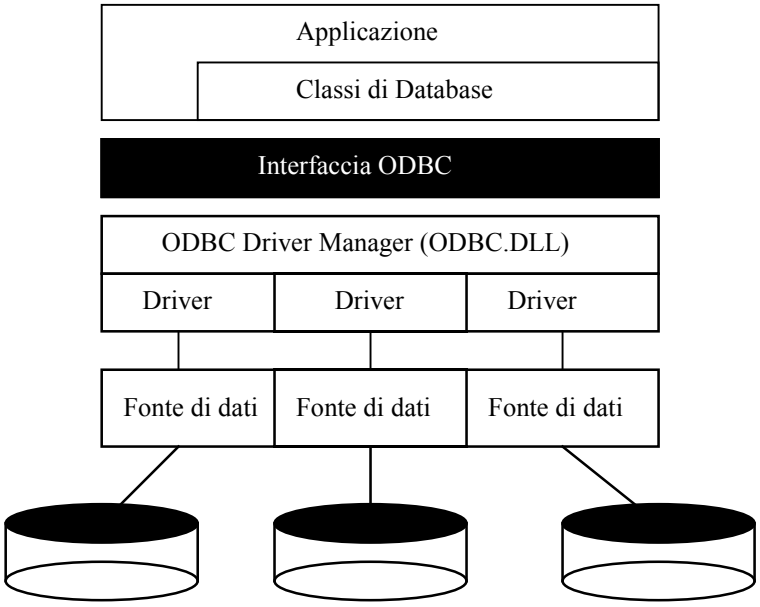
(N.B:questa funzione viene richiamata anche per la visualizzazione a video)
- CDC::StartPage (richiamata per ogni pagina)
- CMyView::OnPrint (richiamata per ogni pagina)
  stampa l'intestazione,pie' di pagina
  richiama OnDraw (se il programmatore vuole una stampa WYSIWYG)
  oppure si preoccupa essa stessa del contenuto della stampa
- CDC::EndPage (qui si itera sulle pagine)
- CDC::EndDoc

- CMyView::OnEndPrinting
  Dealloca le risorge GDI per la stampa

Il framework utilizza una struttura CPrintInfo per mantenere tutte le informazioni di stampa durante il processo. CPrintInfo permette di gestire la numerazione delle pagine, un controllo di fine stampa e una suddivisione del foglio di stampa per avere le intestazioni e i pie' di pagina

ACCESSO ALLA BASE DI DATI TRAMITE ODBC

Il Visual C++ incorpora molti componenti chiave ODBC, tra cui i file di intestazione, le librerie e le DLL richieste. Fornisce anche strumenti, tra cui:
-il Drive Manager di ODBC,
-la Cursor Library di ODBC e
-l'applicazione del pannello di controllo: ODBC Administrator, che viene utilizzata per configurare le fonti di dati.
Inoltre, le classi di database di MFC sono basate su ODBC per assicurare la massima interoperabilità. La figura sottostante mostra le relazioni tra un'applicazione, le classi di database, ODBC e un DBMS.



A questo punto diventa necessario fornire una visione di insieme di Open Database Connectivity (ODBC): è una interfaccia a livello di chiamata che consente alle applicazioni di accedere ai dati in un database qualsiasi per il quale esiste un driver ODBC. Esso, infatti, fornisce un'API che consente ad una applicazione di essere indipendente dal sistema di gestione di database sorgente (DBMS). I driver ODBC gestiscono i collegamenti alla fonte dei dati e si usa SQL per selezionare i record dal database.

I componenti di ODBC sono: API, Driver Manager, driver di database, Cursor Library e Administrator.

Il primo è una libreria di chiamate di funzione, un insieme di codici di errore e una sintassi di Structured Query Language (SQL) standard per accedere ai dati sui DBMS.

Il secondo è una libreria a collegamento dinamico (ODBC32.DLL) che carica i driver di database di ODBC per conto di un'applicazione. Questa DLL è trasparente per la nostra applicazione.

I driver di database consistono in una o più DLL che elaborano le chiamate di funzioni di ODBC per specifici DBMS.

La Cursor Library è una libreria a collegamento dinamico (ODBCCR32.DLL) che risiede tra il Driver Manager di ODBC e i driver handle che scorrono attraverso i dati.

ODBC Administrator, infine, è uno strumento usato per la configurazione di un DBMS in modo tale da renderlo disponibile come fonte di dati per un'applicazione.

PASSI DA SEGUIRE PER UTILIZZARE ODBC:
- creare la fonte dei dati e renderla visibile al sistema windows.

- creare l'applicazione con classwizard, specificando la necessita del supporto ai database
- collegarsi con la fonte dei dati
- appwizard crea una classe derivata dalla CRecordSet, gia' collegata con la fonte dei dati
- ora il programmatore deve solo manipolare i dati ottenuti dalla classe, oppure puo' sfruttare le funzioni membro offerte dalla classe per fare nuove interrogazioni e operazioni sul database.

LA CLASSE CRECORDSET
Attraverso la classe CRecordSet e' possibile accedere:
-a una tabella (es. i CLIENTI)
-ad una interrogazione (SELECT di SQL)
-ad un join di tabelle
-ad una procedura gia' memorizzata nel database (CALL)

Quando si crea una classe derivata dalla CRecordSet, ClassWizard oppure il programmatore manualmente, definisce dentro di essa:
i dati membri relativi ai campi del record
i dati membri dei parametri (per interrogazioni dinamiche)
i dati membri **m_nFields** , **m_nParams** , **m_strFilter**

Quando si effettua la selezione (interrogazione) in esecuzione (Open) o quando ci si muove fra i record (MoveNext,MovePrec ecc...), il framework si occupa automaticamente di mappare il record corrente con i dati membro della classe derivata da CRecordSet.
La selezione puo' essere fissa (ad esempio SELECT INDIRIZZO FROM CLIENTI WHERE COGNOME='ROSSI') oppure puo' essere una interrogazione dinamica, in cui posso cambiare le stringhe da cercare in fase di esecuzione (ad esempio SELECT INDIRIZZO FROM CLIENTI WHERE COGNOME=?). In questo caso si deve inserire un segnaposto ('?') e successivamente in fase di esecuzione si sostituisce il segnaposto con il valore di una variabile stringa (detta parametro).
m_nFields contiene il numero di campi del record (inizializzato da ClassWizard)
m_nParams contiene il numero di parametri della interrogazione (questo deve essere inizializzato manualmente poiche' class wizard non supporta i parametri, e quindi le interrogazioni dinamiche).

Se si utilizza ClassWizard per creare la classe derivata da CRecordSet, esso si connette gia' alla fonte dei dati, inoltre per default prepara gia' una istruzione SQL adattata a cio' che si e' richiesto al momento della creazione della classe. Questo e' il comportamento di default, ma e' possibile in qualsiasi momento cambiarlo. Infatti e' possibile:
-aprire la connessione di default (Open senza parametri)
-aprire la connessione con una istruzione SQL diversa (Open(<istr. SQL>)) utilizzando clausole particolari (ORDER BY,GROUP BY,SUM,MIN,MAX,AVG,COUNT)
-configurare la connessione per la sola lettura
-ecc...

Le operazioni possibili una volta creata la connessione ed eseguita l'istruzione SQL sono:
-scorrere tra i record selezionati (MoveFirst, MoveNext,MovePrec,MoveLast,Move)
-testare l'inizio o la fine della selezione (IsBOF,IsEOF)
-tornare ad eseguire l'interrogazione (Requery)
-aggiungere,eliminare,modificare record

AGGIORNARE,AGGIUNGERE ED ELIMINARE UN RECORD

Per compiere una di queste 3 operazioni prima di tutto bisogna verificare che l'operazione sia permessa, attraverso CanUpdate(aggiornamento ed eliminazione) e CanAppend (aggiunta).
Per aggiungere un record bisogna richiamare AddNew, cosi facendo il framework prepara un record nuovo (vuoto), successivamente lo si riempie (in realta' si riempiono i dati membri relativi ai campi e poi c'e' un meccanismo di mapping di essi sul recod effettivo) e alla fine si richiama Update per effettuare la scrittura.
Per modificare un record bisogna posizionarsi su di esso, richiamare Edit, modificare i dati membri relativi ai campi, alla fine si richiama Update per effettuare la scrittura delle modifiche.
Per eliminare un record bisogna posizionarsi su di esso,richiamare Delete, e questo basta per marcarlo come cancellato. Attenzione che dopo una chiamata Delete non si e' piu' posizionati su alcun record quindi prima di fare altre operazioni bisogna riposizionarsi (es MoveNext).

COME UN RECORDSET COSTRUISCE LA ISTR. SQL

Sono diverse le possibilita' offerte dalla classe CRecordSet per costruire l'istruzione SQL. Prima di tutto il framework controlla se voi programmatori avete specificato una istruzione SQL nella Open, al che esso utilizza quella istruzione che ha la precedenza assoluta. Inoltre voi potete passare ad Open solamente cio' che sta dopo FROM nella istr. SQL. Se la Open non ha specificato niente si procede nel seguente modo:

SELECT <membri di dato(campi)> FROM GetDefaultSQL WHERE m_strFilter
ORDER BY m_strSort

Il framework e' gia' in grado di mappare i dati membro con i nomi dei campi del record.
GetDefaultSQL restituisce il nome della tabella
m_strFilter e' un dato membro stringa contenente la condizione (con eventuali '?')
m_strSort e' un dato membro stringa contenente il/i campo/i da ordinare

Dopo aver costruito l'istruzione, Open invia la istruzione stessa al driver manager di ODBC, che la manda allo specifico driver odbc per lo specifico dbms. Il driver (E SOLO LUI) e' in grado di convertire l'istruzione in codice appropriato per accedere al particolare formato delle tabelle, e quindi restituire al framework i corretti record.

OSSERVAZIONE : moltissime sono le possibilita' offerte dai driver ODBC, purtroppo molto meno sono le possibilita' offerte da ClassWizard e dalla classe CRecordSet. Se serve qualcosa di specifico non supportato da ClassWizard (o dalla classe CRecordSet) bisogna accedere direttamente alle API ODBC di Windows

PARAMETRIZZARE UN RECORDSET

Per fare interrogazioni con condizioni articolate e mutevoli durante l'esecuzione del programma, ci sono varie possibilita'. La piu' veloce ed elegante e' quella di utilizzare i parametri. Questa parametrizzazione non e' supportata (per ora) da ClassWizard e quindi bisogna intervenire direttamente sul codice.

Per usare un parametro nella stringa m_strFilter ci deve essere almeno un segnaposto (indicato con '?'). Ad esempio m_strFilter="NOME = ?".

Supponete di ottenere in fase di esecuzione il nome del cliente da cercare nella variabile nuovocli, variabile di tipo stringa. Supponete di avere nella variabile nuovocli=“Rossi”. Adesso indico le fasi da seguire per parametrizzare la classe con questa variabile nuovocli.
1) eseguire ClassWizard e creare la classe derivata dalla CRecordSet agganciandosi alla tabella clienti, creando tutti i dati membri relativi ai campi che si vogliono ottenere dalla interrogazione.
2) aggiungere manualmente nel file .H il dato membro relativo al parametro (si consiglia di metterlo fuori dallo spazio riservato a ClassWizard). Ad es:

```
// Field/Param Data
        //{{AFX_FIELD(CEsSet, CRecordset)
        CString   m_codice;
        CString   m_nome;
        CString   m_indirizzo;
        CString   m_telefono;
        //}}AFX_FIELD
        Cstring  m_str1Param;          //in genere per convenzione si finisce con Param
```

3) Si introduce manualmente nella implementazione della funzione DoFieldExchange (che si incarica di mappare i campi con le variabili membro) una voce di scambio dati (RFX). Ad es:

```
        //{{AFX_FIELD_MAP(CEsSet)
        pFX->SetFieldType(CFieldExchange::outputColumn);
        RFX_Text(pFX, "CODICE CLIENTE", m_codice);
        RFX_Text(pFX, "NOME", m_nome);
        RFX_Text(pFX, "INDIRIZZO", m_indirizzo);
        RFX_Text(pFX, "TEL", m_telefono);
        //}}AFX_FIELD_MAP
        pFX->SetFieldType(CFieldExchange::param);
        RFX_Text(pFX, "PARAM1", m_str1Param);
```

4) Si inserisce nel costruttore della classe derivata da CRecordSet nel dato membro m_nParams il numero di parametri (in questo caso 1)

```
        //{{AFX_FIELD_INIT(CEsSet)
        m_codice = "";
        ...
        //}}AFX_FIELD_INIT
        m_nParams = 1;
```

5) nel momento della creazione dell’oggetto derivato da CRecordSet (o prima della chiamata di Open) bisogna inserire il segnaposto (o i segnaposto) in m_strFilter

```
        m_strFilter=“NOME = ?”;
```

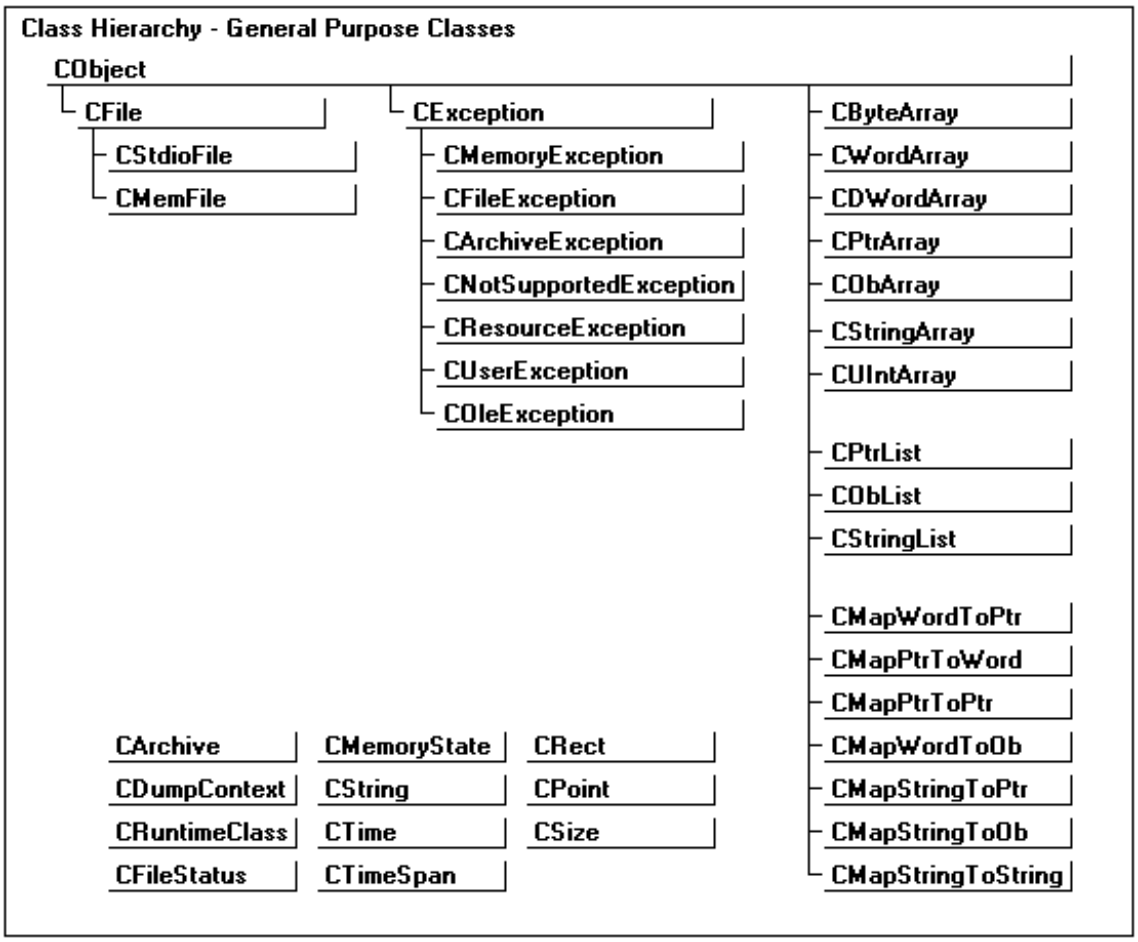
In fase di esecuzione il framework sostituisce i segnaposto con i parametri che sono stati specificati dopo la chiamata **pFX->SetFieldType(CFieldExchange::param);** in DoFieldEchange, e per fare questo e solo l’ordine che mappa i segnaposto con i nomi dei parametri.

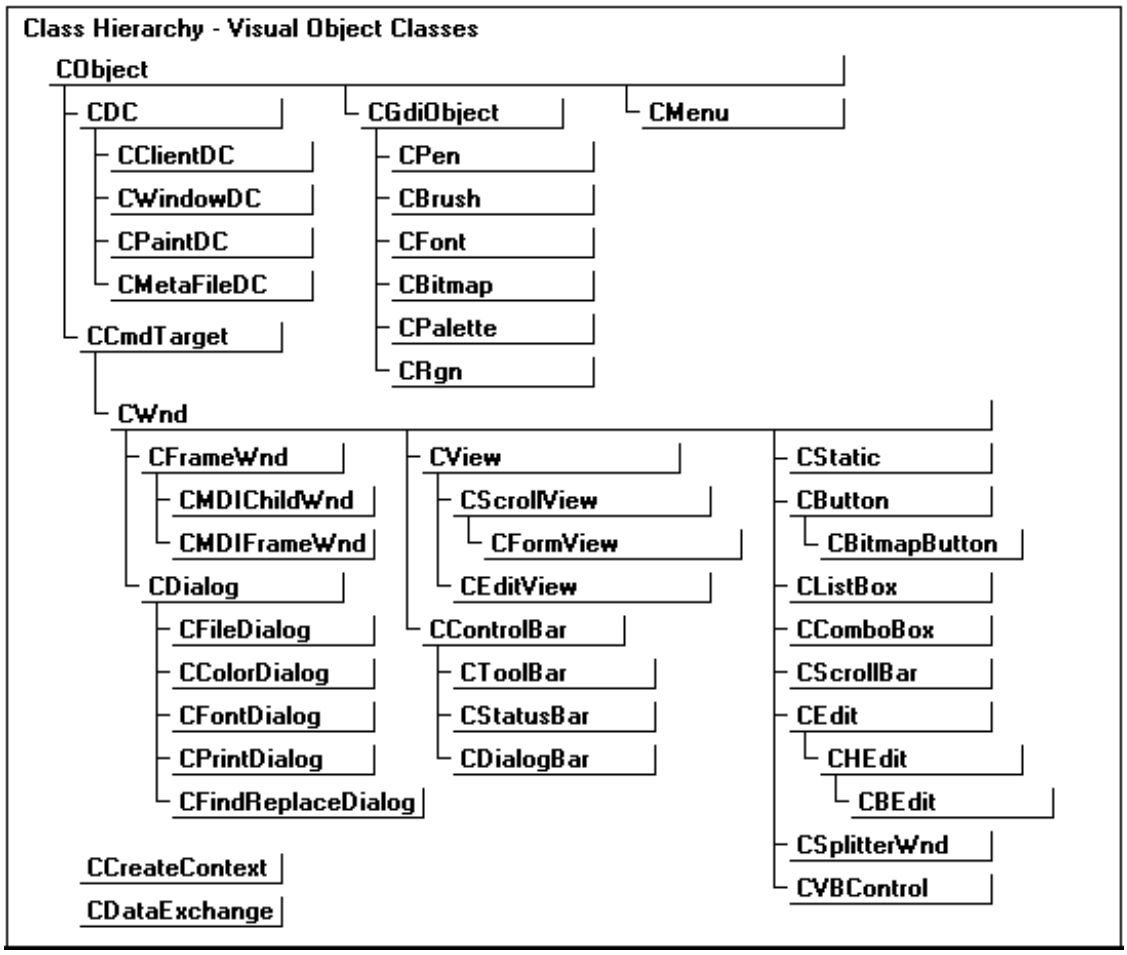
6) A questo punto e' possibile far partire qualsiasi interrogazione basata sul contenuto di nuovocli. Ad es:

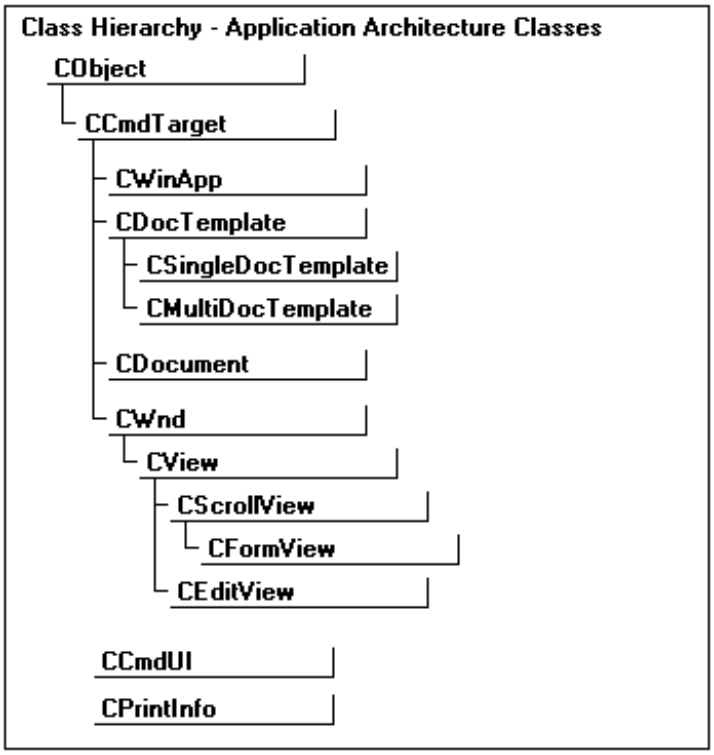
```
<nome istanza>.m_str1Param=nuovocli;
...
<nome istanza>.Open();
```

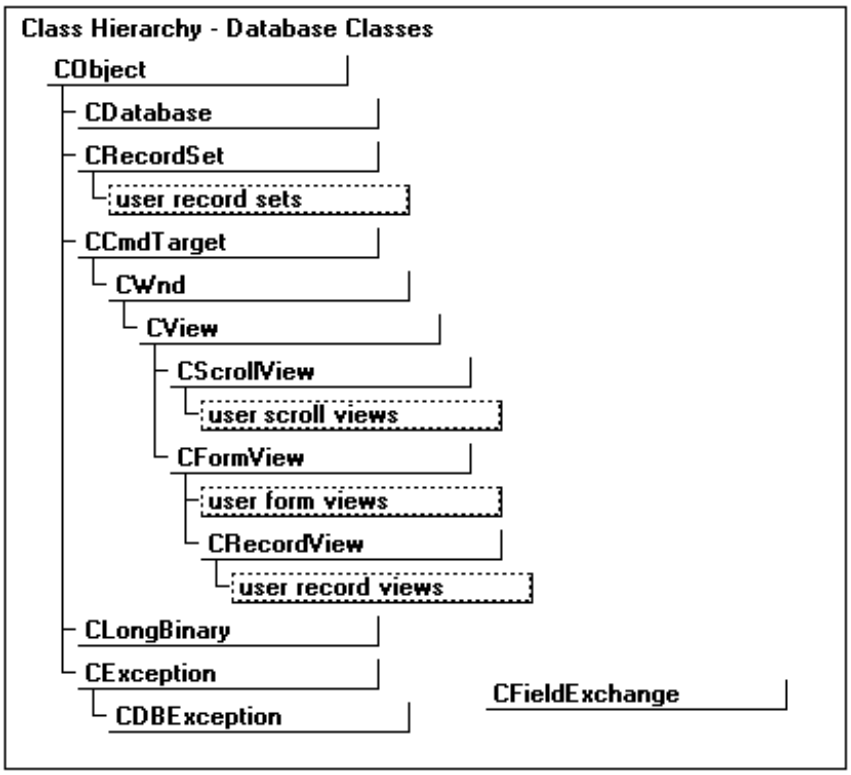
In questo caso se nuovocli="Rossi", tutta questa complicata operazione esegue l'istruzione SQL : SELECT ... FROM CLIENTI WHERE NOME = 'Rossi'.
In un secondo momento in fase di esecuzione dentro a nuovocli ci potrebbe essere "Bianchi" e tutto funziona correttamente, restituendomi tutti i clienti di nome Bianchi.

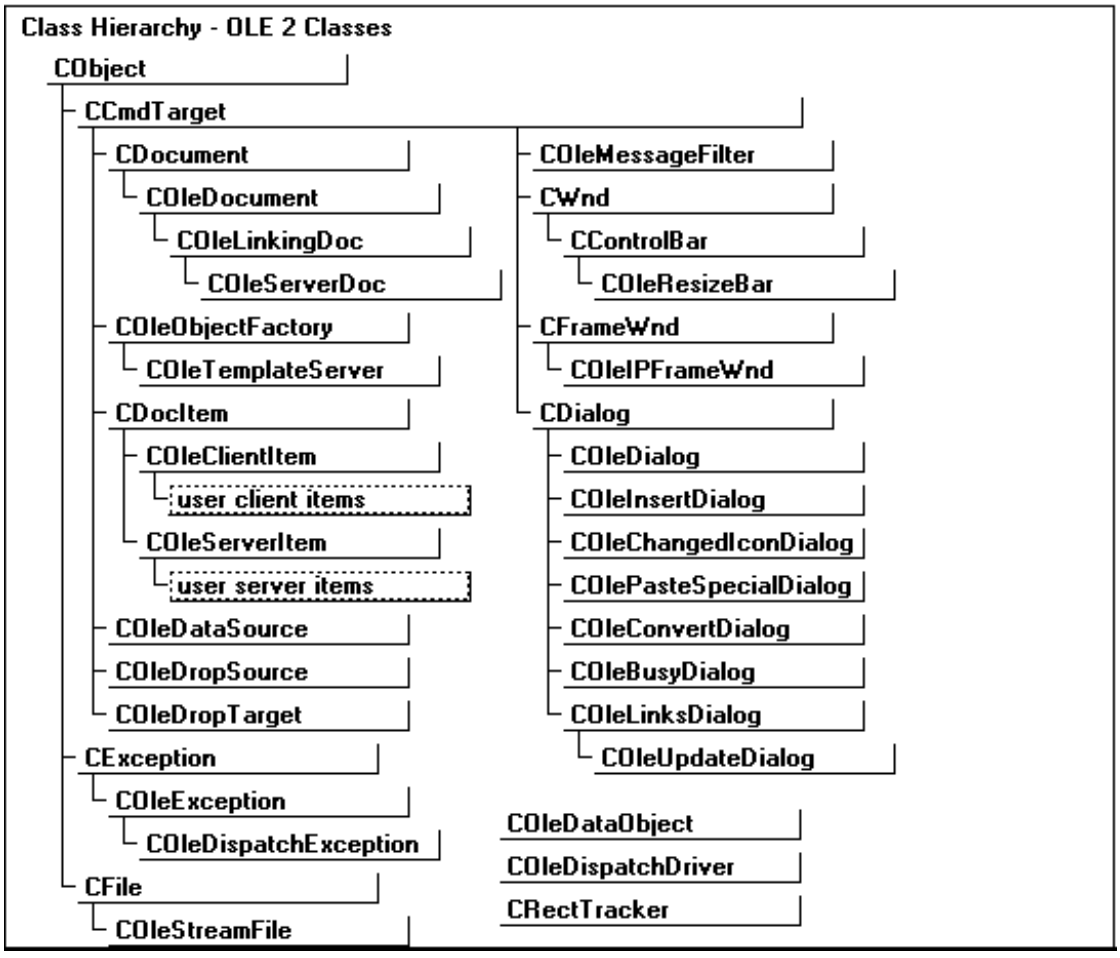
APPENDICE – GERARACHIA DI CLASSI MFC




Class Hierarchy - Visual Object Classes
CObject
CDC
CClientDC
CWindowDC
CPaintDC
CMetaFileDC
CGdiObject
CPen
CBrush
CFont
CBitmap
CPalette
CRgn
CMenu
CCmdTarget
CWnd
CFrameWnd
CMDIChildWnd
CMDIFrameWnd
CDialog
CFileDialog
CColorDialog
CFontDialog
CPrintDialog
CFindReplaceDialog
CView
CScrollView
CFormView
CEditView
CControlBar
CToolBar
CStatusBar
CDialogBar
CStatic
CButton
CBitmapButton
CListBox
CComboBox
CScrollBar
CEdit
CHEdit
CBEdit
CSplitterWnd
CVBControl
CCreateContext
CDataExchange


Class Hierarchy - Application Architecture Classes
CObject
CCmdTarget
CWinApp
CDocTemplate
CSingleDocTemplate
CMultiDocTemplate
CDocument
CWnd
CView
CScrollView
CFormView
CEditView
CCmdUI
CPrintInfo


Class Hierarchy - Database Classes
CObject
CDatabase
CRecordSet
user record sets
CCmdTarget
CWnd
CView
CScrollView
user scroll views
CFormView
user form views
CRecordView
user record views
CLongBinary
CException
CDBException
CFieldExchange


Class Hierarchy - OLE 2 Classes
CObject
CCmdTarget
CDocument
COleDocument
COleLinkingDoc
COleServerDoc
COleObjectFactory
COleTemplateServer
CDocItem
COleClientItem
user client items
COleServerItem
user server items
COleDataSource
COleDropSource
COleDropTarget
CException
COleException
COleDispatchException
CFile
COleStreamFile
COleMessageFilter
CWnd
CControlBar
COleResizeBar
CFrameWnd
COleIPFrameWnd
CDialog
COleDialog
COleInsertDialog
COleChangedIconDialog
COlePasteSpecialDialog
COleConvertDialog
COleBusyDialog
COleLinksDialog
COleUpdateDialog
COleDataObject
COleDispatchDriver
CRectTracker