«Ribbon» e riga di stato

di Sergio Polini

Antonio Lombrano, di Massa, mi chiede di trattare un problema di sicuro interesse: come visualizzare in un programma una schermata grafica creata con Paintbrush, ma anche, più in generale, immagini memorizzate in file PCX, TIFF, ecc. Non è escluso che, in un prossimo futuro, possa essere accontentato. Per il momento, segnalo che in un libro di Martin Heller, Advanced Windows Programming, *edito dalla John Wiley, viene dedicato ampio spazio a tali argomenti. Mentre scrivo il libro non è ancora uscito, ma con ogni probabilità sarà disponibile quando leggerete queste pagine*

La volta scorsa abbiamo visto come sia necessario, a volte, ricorrere alle funzioni della API pur programmando in ObjectWindows, accennando però anche alla possibilità di sfruttare le possibilità della OOP per creare una gerarchia di classi facilmente utilizzabile in diverse applicazioni. Ora approfondiremo soprattutto quest'ultimo aspetto, senza dimenticare il nostro obiettivo: creare schemi di applicazioni facilmente portabili da Turbo Vision a ObjectWindows e viceversa.

Dopo aver visto come cambiare il menu di un'applicazione MDI (cosa che faremo anche in Turbo Vision), si tratta ora di implementare una riga di stato; dovremo riuscire a mostrare in essa una descrizione delle opzioni dei menu man mano che l'utente le passa in rassegna con la tastiera o con il mouse, ma dovremo anche implementare un equivalente sotto Windows di un particolare aspetto delle righe di stato di un'applicazione Turbo Vision; mi riferisco alla possibilità di attivare alcuni comandi non solo premendo i tasti ricordati nella riga di stato, ma anche «clickando» su questa col mouse. In realtà sotto Windows si vedono, di norma, solo righe di stato «passive», nel senso che mostrano informazioni di vario genere ma non reagiscono ai click del mouse; a tale scopo si usano piuttosto i cosiddetti *ribbon*, ovvero «righe» poste subito sotto il menu e ricche di combo box, pulsanti, ecc. Potremmo dire, quindi, che una riga di stato Turbo Vision comprende in sé le funzionalità sia di una riga di stato che di un ribbon, funzionalità che conviene tenere separate quando si lavora sotto Windows.

La numerazione dei messaggi

Riga di stato e ribbon sono ambedue «righe», ma sarebbe meglio dire «stri-

Figura 1 - La numerazione (in esadecimale) dei messaggi di Windows come documentata nei manuali dell'SDK.

```
0000 - 03FF:  messaggi di Windows
0400 - 7FFF:  messaggi per le applicazioni
8000 - BFFF:  riservati per future versioni di Windows
C000 - FFFF:  messaggi creati con RegisterWindowMessage
```

```
            0000  - 03FF:  messaggi di Windows
wm_User     (0400) - 7FFF:  messaggi per le applicazioni
                            (wm_User + [0..7BFF];
                            in decimale: wm_User + [0..31743])
id_First    (8000) - 8EFF:  messaggi generati dai controlli
                            (id_First + [0..0EFF];
                            in decimale: id_First + [0..3839])
            8F00  - 8FFF:   riservati
nf_First    (9000) - 9EFF:  messaggi di notifica dai controlli
                            (nf_First + [0..0EFF];
                            in decimale: nf_First + [0..3839])
            9F00  - 9FFF:   riservati
cm_First    (A000) - FEFF:  messaggi da menù e acceleratori
                            (cm_First + [0..5EFF];
                            in decimale: cm_First + [0..24319])
            FF00  - FFFF:   riservati
```

Figura 2 - La numerazione (in esadecimale) dei messaggi di Windows come organizzata in ObjectWindows.

```
unit Stripes;

interface

uses
  WinTypes, WinProcs, WObjects;

const

  um_MDI              = 28672; (* $7000 *)
  um_MDIChildDestroy  = um_MDI;
  um_SetActiveChild   = um_MDI + 1;
  um_SetHelpCode      = um_MDI + 2;
  um_PaintStatusBar   = um_MDI + 3;

  cm_ToggleRibbon     = 24316; (* $5EFC *)
  cm_ToggleStatusBar  = 24317; (* $5EFD *)
  cm_ActivateChild    = 24318; (* $5EFE *)
  cm_ChildList        = 24319; (* $5EFF *)

  idf_MainText        = 101;

  ids_PopupMenu       = 1000;
  ids_MenuItem        = 2000;
  ids_ChildPopupMenu  = 3000;
  ids_ChildMenuItem   = 4000;

type

  PStripe = ^TStripe;
  TStripe = object(TDialog)
    Height : Integer;
    Visible: Boolean;
    constructor Init(AParent: PWindowsObject; AName: PChar;
                     IsVisible: Boolean);
    procedure SetupWindow; virtual;
  end;

  PMDIRibbon = ^TMDIRibbon;
  TMDIRibbon = object(TStripe)
    procedure WMEnable(var Msg: TMessage);
      virtual wm_First + wm_Enable;
  end;

  PMDIStatusBar = ^TMDIStatusBar;
  TMDIStatusBar = object(TStripe)
    MainText: PStatic;
    constructor Init(AParent: PWindowsObject; AName: PChar;
                     IsVisible: Boolean);
    procedure PaintField(Field: Integer; Text: PChar); virtual;
  end;

implementation

constructor TStripe.Init(AParent: PWindowsObject; AName: PChar;
                         IsVisible: Boolean);
begin
  TDialog.Init(AParent, AName);
  EnableAutoCreate;
  Visible := IsVisible;
end;

procedure TStripe.SetupWindow;
var
  R: TRect;
begin
  TDialog.SetupWindow;
  if Visible then Show(sw_ShowNoActivate);
  GetClientRect(HWindow, R);
  Height := R.Bottom - R.Top + 2;
  EnableWindow(HWindow, False);
end;

procedure TMDIRibbon.WMEnable(var Msg: TMessage);
  procedure Enable(P: PWindowsObject); far;
  begin
    EnableWindow(P^.HWindow, Bool(Msg.WParam));
  end;
begin
  ForEach(@Enable);
end;

constructor TMDIStatusBar.Init(AParent: PWindowsObject; AName: PChar;
IsVisible: Boolean);
begin
  TStripe.Init(AParent, AName, IsVisible);
  MainText := New(PStatic, InitResource(@Self, idf_MainText, 100));
end;

procedure TMDIStatusBar.PaintField(Field: Integer; Text: PChar);
begin
  SendDlgItemMsg(Field, wm_SetText, 0, Longint(Text));
end;

end.
```

Figura 3 - La unit STRIPES, che implementa le funzionalità fondamentali di un ribbon e di una riga di stato.

sce», in quanto possono essere implementate anche graficamente. Partiremo quindi da una unit STRIPES e da una classe *TStripe*, nella quale raccoglieremo quanto hanno in comune un ribbon e una riga di stato.

Prima, però, dobbiamo dare un'occhiata alla numerazione dei messaggi di Windows. Questa è organizzata in modo piuttosto semplice, come si può constatare osservando la figura 1. ObjectWindows propone una diversa articolazione, illustrata nella figura 2; si può notare, tra l'altro, che vengono invasi sia il range riservato per future versioni di Windows, sia quello destinato ai messaggi installati mediante la funzione *RegisterWindowMessage* (utile per definire messaggi comuni a diverse applicazioni). Soprassediamo ora dalle implicazioni di tale sovrapposizione, la cui analisi ci spingerebbe troppo lontano, fino ai più nascosti dettagli dell'implementazione di ObjectWindows; preoccupiamoci piuttosto di evitare altre sovrapposizioni, quelle che potrebbero capitare definendo costanti per messaggi e co-

```
unit MDIClnt;

interface

uses WinTypes, WinProcs, WObjects, Stripes;

type
  PNewMDIClient = ^TNewMDIClient;
  TNewMDIClient = object(TMDIClient)
    procedure UMMDIChildDestroy(var Msg: TMessage);
      virtual wm_First + um_MDIChildDestroy;
  end;

implementation

procedure TNewMDIClient.UMMDIChildDestroy(var Msg: TMessage);
var
  HFrameMenu, HWinMenu: HMenu;
begin
  if LoWord(SendMEssage(HWindow, wm_MDIGetActive, 0, 0)) = 0 then begin
    HFRameMenu := PMDIWindow(Parent)^.Attr.Menu;
    HWinMenu := GetSubMenu(HFrameMenu, 0);
    SendMessage(HWindow, wm_MDISetMenu,
                0, MakeLong(HFrameMenu, HWinMenu));
    DrawMenuBar(Parent^.HWindow);
    SendMessage(Parent^.HWindow, um_SetActiveChild, 0, Longint(nil));
    SendMessage(Parent^.HWindow, um_PaintStatusBar, idf_MainText, 0);
  end;
end;

end.
```

Figura 4 - La unit MDICLNT, che implementa una client window *capace di gestire non solo menu variabili (come già visto il mese scorso), ma anche ribbon e riga di stato.*

A proposito di InitResource...

Su MC-link un abbonato ha chiesto: se creo con il Whitewater Resource Toolkit una dialog box con un controllo di tipo *edit*, quando si esegue *New(PEdit, InitResource(...))* si crea una seconda istanza di *TEdit*? Mi ha dato così l'occasione di chiarire un aspetto fondamentale di ObjectWindows (chi voglia rileggere la discussione e il suo contesto, può cominciare dai messaggi 805 e 809 dell'area PASCAL di MC-link). Una delle cose più importanti da ricordare quando si usa ObjectWindows, infatti, è la differenza tra *oggetti d'interfaccia* e *elementi d'interfaccia*. Un oggetto d'interfaccia è una istanza di una classe di ObjectWindows, che si limita a fornire gli attributi e il comportamento di ciò che vediamo sullo schermo, mentre l'implementazione «fisica», l'apparenza visiva dell'oggetto è fornita dal corrispondente elemento d'interfaccia, cioè dall'«oggetto» tratto dal bagaglio di Windows.

Come ho già avuto modo di dire, in Windows troviamo un'architettura concettualmente orientata all'oggetto ed una implementazione tradizionale. ObjectWindows restituisce alla programmazione sotto Windows una maggiore coerenza tra disegno e implementazione, ma... non riscrive Windows! Gli oggetti della gerarchia di classi, quindi, vengono visualizzati attingendo al ricco bagaglio che Windows comunque mette a disposizione. Ne segue che un oggetto d'interfaccia, per apparire sullo schermo, deve legarsi ad un elemento d'interfaccia fornito da Windows.

Nel caso di una finestra, il legame è rappresentato dal campo *HWindow* dell'istanza di *TWindow* (l'oggetto d'interfaccia), in cui va l'*handle* della finestra creata da Windows (l'elemento d'interfaccia). L'oggetto nella sua interezza è completo solo dopo che il legame è stato stabilito. È questo il motivo per cui, ad esempio, non posso usare in *TApplication.InitMainWindow* funzioni della API che vogliano *HWindow* come parametro, in quanto lì viene chiamato un constructor che costruisce solo l'oggetto d'interfaccia. Si deve aspettare che sia stato eseguito *TApplication.InitInstance*, che, dopo aver chiamato *InitMainWindow*, costruisce l'elemento d'interfaccia chiamando *MakeWindow* (che a sua volta chiama *MainWindow^.Create*). Analogo il caso delle dialog box, anche se queste vengono tratte da un file di risorse: l'oggetto è completo solo dopo la costruzione dell'elemento d'interfaccia mediante *ExecDialog* o *MakeWindow*, oppure, se si usa *EnableAutoCreate* (come in STRIPES.PAS), durante il metodo *SetupWindow* della finestra cui la dialog box appartiene.

Nel caso dei controlli (*button*, edit, ecc.), si deve distinguere tra quelli posti in una finestra «normale», per i quali vi è un'analoga associazione tra oggetto e elemento d'interfaccia (creato dal metodo *SetupWindow*), e quelli posti in una dialog box definita in un file di risorse. Questi ultimi non sono oggetti d'interfaccia, in quanto sono gestiti in tutto e per tutto dalla dialog box cui appartengono. Sono elementi d'interfaccia il cui comportamento è definito e gestito da Windows. Il legame oggetto-elemento è indiretto, mediato dalla dialog box: il click del mouse su un controllo genera un messaggio cui risponde un metodo della dialog box, non del controllo.

È tuttavia possibile associare un *oggetto di controllo* ad un *elemento di controllo* di una dialog box. Questo si fa definendo l'oggetto di controllo come istanza di una classe derivata da *TControl*, e poi costruendo l'oggetto di controllo nell'ambito del constructor della dialog box, ma usando per lui *InitResource* invece di *Init*.

Il risultato sarà che il controllo cambierà comportamento: invece di avere il comportamento che Windows definisce per esso (in quanto elemento di controllo), avrà il comportamento definito dalla sua classe. Ma per il resto sarà sempre un unico «oggetto»; invece che essere costruito nelle due componenti oggetto d'interfaccia (creato da Init) e elemento d'interfaccia (creato da *MakeWindow* o *ExecDialog* o *SetupWindow*), verrà costruito mediante *InitResource*, che crea l'oggetto d'interfaccia e lo associa ad un elemento d'interfaccia che c'è già, tanto che *InitResource* richiede anche un parametro avvalorato con il numero che identifica l'elemento d'interfaccia (il cosiddetto id del controllo).

mandi in programmi che utilizzino le unit che vi sto proponendo. A questo scopo, definiremo una costante *um_MDI* con valore pari a $7000 (28672), prossimo al massimo valore possibile, come base per i messaggi definiti nelle unit; useremo inoltre i valori da $5EFC a $5EFF (da 24316 a 24319) per i quattro comandi usati dalle unit.

Strisce generiche

Tutte le costanti comuni alle unit vengono definite nella unit STRIPES (figura 3). In essa viene definita una classe TStripe con le variabili d'istanza *Height* e *Visible*, nonché un constructor e una procedura *SetupWindow*. Sarà necessario descrivere in un file di risorse la struttura del ribbon e della riga di stato, implementati come dialog box non modali dotate di un bordo; la chiamata della procedura *EnableAutoCreate* nel constructor fa si che le nostre strisce vengano create automaticamente insieme alla finestra principale. La variabile *Visible*, avvalorata mediante un parametro del constructor, determina la iniziale visibilità o meno della striscia, mentre *Height* serve a tenere memoria della sua altezza (compreso il bordo), per i motivi che vedremo.

La classe *TMDIRibbon*, capostipite di qualsiasi ribbon, prevede il solo metodo *WMEnable*, destinato a rispondere al messaggio *WM_ENABLE*; la *frame window*, infatti, provvederà ad abilitare o disabilitare il ribbon mediante la funzione API *EnableWindow*, la quale provoca appunto l'invio del messaggio WM_ENABLE. Il corrispondente metodo del ribbon non fa altro che girare il messaggio a tutte le sue *child window* (tipicamente, ai suoi controlli), mediante l'iteratore *ForEach*.

Se un ribbon può essere considerato come una dialog box ricca di controlli della più varia specie, una riga di stato è in genere la sede per controlli di tipo *static*, come una sorta di record di campi di tipo stringa; uno di questi sarà destinato alla visualizzazione di brevi descrizioni delle opzioni dei menu. La classe *TMDIStatusBar* ha quindi una variabile d'istanza *MainText* di tipo *PStatic*, che viene inizializzata nel constructor mediante un *New* con *InitResource*; ciò richiede che il file di risorse comprenda una riga di stato con almeno un campo static, identificato da un numero uguale alla costante *idf_MainText*. Il metodo *PaintField* consente di scrivere un qualsiasi testo in un qualsiasi «campo» della riga di stato; le unit che stiamo esaminando usano solo il campo *idf_MainText*, ma rimane aperta la possibilità di derivare da *TMDIStatusBar* righe di stato con più campi, senza bisogno di modificare la unit STRIPES.

Client window e Child window

Nella figura 4 trovate la unit MDICLNT, in cui si definisce una classe *TNewMDIClient*. Questa è molto simile alla classe *TMyMDIClient* vista il mese scorso; anche qui troviamo un solo metodo, destinato a rispondere al messaggio *um_MDIChildDestroy*. Lo scopo è quello di verificare, quando viene chiusa una *child window*, se si trattava dell'unica che restava aperta. In questo caso, oltre al cambio di menu già visto il mese scorso, si provvede ad inviare alla *frame window* i messaggi *um_SetActiveChild*, con un *LParam* uguale a **nil**, e *um_PaintStatusBar*, con *WParam* uguale a *idf_MainText* e *LParam* uguale a zero; il primo messaggio fa si che venga assegnato **nil** alla variabile d'istanza *ActiveChild* della *frame window*, mentre il secondo provoca la riscrittura della riga di stato.

Troviamo qualcosa di analogo anche nella unit MDICHLD, in cui si definisce una classe *TNewMDIChild*. A differenza di *TNewMDIClient*, da usare così com'è, questa classe va intesa come quella da cui derivare (invece che da *TWindow*) le classi per le vere finestre di un'applicazione, per le quali predispo-

ne le funzionalità fondamentali. Nella figura 5 potete trovare il sorgente della unit, dal quale ho tolto momentaneamente l'implementazione dei metodi *WMMenuSelect* e *WMEnterIdle*, che vedremo il mese prossimo insieme agli omonimi metodi della *frame window*.

In *TNewMDIChild* troviamo una variabile d'istanza *SBText*, destinata a contenere un puntatore al testo da mostrare nel campo *idf_MainText* della riga di stato; la variabile va inizializzata in un constructor che, per il resto, è identico a quello visto il mese scorso. In MDICHLD.PAS si usa la stringa «Child Window» al solo scopo di verificare, nel demo che vi proporrò al termine dell'esposizione, che i vari meccanismi funzionano. Una stringa nulla andrebbe benissimo, in quanto in fondo si tratta di una classe astratta.

I metodi *WMMDIActivate* e *WMDestroy* sono analoghi a quelli già visti a dicembre, con la differenza che nel primo si invia alla *frame window* il messaggio *um_SetActiveChild*, con l'indirizzo dell'istanza della finestra appena attivata in *LParam*, e alla finestra stessa il messaggio *um_PaintStatusBar*, con *idf_MainText* in *WParam* e zero in *LParam*. Il metodo *UMPaintStatusBar* rimanda lo stesso messaggio alla *frame window*; se però *WParam* è uguale a *idf_MainText* e *LParam* vale zero, assegna prima a quest'ultimo *SBText*.

Flessibilità

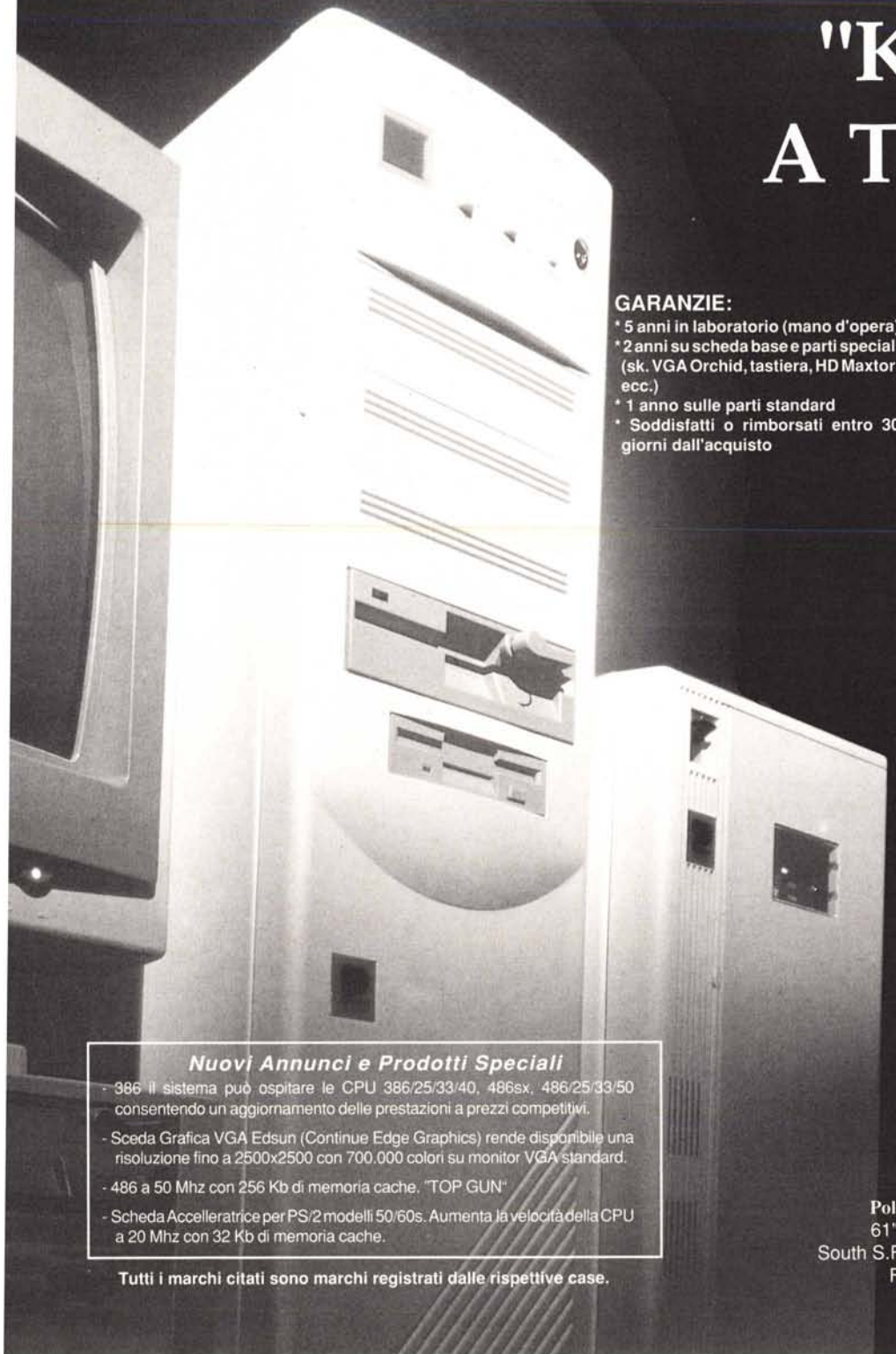
Il meccanismo appena descritto può magari apparire complicato, ma ho cercato di renderlo quanto più possibile flessibile. Lo scopo finale è quello di consentire ad una *child window* di riprodurre nella riga di stato, dopo che su essa è stata mostrata la descrizione di un'opzione di un menu, le stesse informazioni che vi sarebbero state senza questa sorta di interruzione. Potrebbe magari trattarsi di informazioni da aggiornare e ricostruire; in questo caso, se in *WMMDIActivate* non si usasse uno zero in *LParam* (con evidente significato convenzionale), potrebbe risultare necessario ridefinire un metodo troppo delicato nelle classi derivate. Sarà invece sufficiente ridefinire *UMPaintStatusBar*.

Ancora: si potrebbe pensare che è inutile che la *child window* mandi un messaggio a se stessa e risponda a questo con un metodo dinamico; potrebbe più semplicemente chiamare un normale metodo virtuale. In realtà, tuttavia, vedremo che anche la *frame window* può mandare quello stesso messaggio ad una *child window*; se, per ottenere lo scopo, si usassero normali metodi virtuali, ciò comporterebbe che la classe della *frame window* dovrebbe «conoscere» quella della *child window*. Nessun problema se la *child window* fosse sempre derivata da *TNewMDIChild*, ma ciò renderebbe complicato l'uso di classi già disponibili, come quelle presenti nella unit STDWNDS.PAS (ad esempio *TFileWindow*). La unit MDIFRAME, che vedremo il mese prossimo, è quindi disegnata in modo da poter prescindere dalla unit MDICHLD; questa può quindi anche essere intesa come un insieme dei frammenti di codice da aggiungere ad un'altra classe, la quale sia derivata da una qualsiasi altra in una unit che, per essere integrata nello schema che stiamo costruendo, potrà limitarsi ad «usare» le unit STRIPES e MDICLNT. MC

Sergio Polini è raggiungibile tramite MC-link alla casella MC1166.

```
unit MDIChld;

interface

uses WinTypes, WinProcs, WObjects, Stripes, MDIClnt;

type
  PNewMDIChild = ^TNewMDIChild;
  TNewMDIChild = object(TWindow)
    SBText: PChar;
    constructor Init(AParent: PWindowsObject; ATitle, AMenu: PChar);
    procedure WMMDIActivate(var Msg: TMessage);
      virtual wm_First + wm_MDIActivate;
    procedure WMDestroy(var Msg: TMessage);
      virtual wm_First + wm_Destroy;
    procedure WMMenuSelect(var Msg: TMessage);
      virtual wm_First + wm_MenuSelect;
    procedure WMEnterIdle(var Msg: TMessage);
      virtual wm_First + wm_EnterIdle;
    procedure UMPaintStatusBar(var Msg: TMessage);
      virtual wm_First + um_PaintStatusBar;
  end;

implementation

var
  HChildMenu: HMenu;

constructor TNewMDIChild.Init(AParent: PWindowsObject;
                              ATitle, AMenu: PChar);
begin
  TWindow.Init(AParent, ATitle);
  SBText := 'Child Window';  (* NB: Solo per demo! *)
  if HChildMenu = 0 then
    HChildMenu := LoadMenu(HInstance, AMenu);
  Attr.Menu := HChildMenu;
end;

procedure TNewMDIChild.UMPaintStatusBar(var Msg: TMessage);
begin
  if (Msg.LParam = 0) and (Msg.WParam = idf_MainText) then
    Msg.LParam := Longint(SBText);
  SendMessage(Parent^.HWindow, um_PaintStatusBar,
              Msg.WParam, Msg.LParam);
end;

procedure TNewMDIChild.WMMDIActivate(var Msg: TMessage);
var
  HWinMenu: HMenu;
  n       : Word;
  Found   : Boolean;
begin
  if Msg.WParam <> 0 then begin
    n := GetMenuItemCount(Attr.Menu);
    Found := FALSE;
    while (n > 0) and (not Found) do begin
      HWinMenu := GetSubMenu(Attr.Menu, n - 1);
      if GetMenuState(HWinMenu, cm_TileChildren, mf_ByCommand) <> $FFFF
      then
        Found := TRUE;
      Dec(n);
    end;
    SendMessage(PMDIWindow(Parent)^.ClientWnd^.HWindow, wm_MDISetMenu,
                0, MakeLong(Attr.Menu, HWinMenu));
    DrawMenuBar(Parent^.HWindow);
    SendMessage(Parent^.HWindow, um_SetActiveChild, 0, Longint(@Self));
    SendMesse e(HWindow, um_PaintStatusBar, Idf_MainText, 0);
  end;
end;

procedure TNewMDIChild.WMDestroy(var Msg: TMessage);
begin
  PostMessage(PMDIWindow(Parent)^.ClientWnd^.HWindow,
              um_MDIChildDestroy, 0, 0);
  TWindow.WMDestroy(Msg);
end;

(* L'implementazione dei metodi WMMenuSelect e WMEnterIdle *)
(* verra' mostrata il mese prossimo                        *)

begin
  HChildMenu := 0;
end.
```

Figura 5 - La unit MDICHLD, che implementa la classe TNewMDIChild, *da usare, in luogo di* TWindow, *per la derivazione delle classi per le finestre dell'applicazione.*



"Killer Price And A Terrific Warranty"

PC Word - giugno 1990

Direttamente dagli USA

GARANZIE:
* 5 anni in laboratorio (mano d'opera)
* 2 anni su scheda base e parti speciali (sk. VGA Orchid, tastiera, HD Maxtor, ecc.)
* 1 anno sulle parti standard
* Soddisfatti o rimborsati entro 30 giorni dall'acquisto

PC MAGAZINE EDITORS' CHOICE

INFO WORLD TOP 100

"Excellent value"
"An impressive package"

Poly 386 - 25/Cache
* 2 MB RAM
* 80 MB Hard Disk
* Sk. graf.VGA
* Monitor Mono 640x480 VGA
* contenitore tower

Lire 2.850.000

INFO WORLD

"Compared with the top 18 manufacturers, Polywell ranked fastest and lowest priced..."

Poly 386sx - 25
* 1MB RAM
* 40MB Hard Disk
* Sk. graf.VGA
* Monitor Mono 640x480 VGA

Lire 2.050.000

PC MAGAZINE

BUYERS ASSURANCE SEAL INFO WORLD

"Shines in its attention to detail."

Poly 386 - 33/Cache
* 64K Cache
* 4MB RAM
* 120 MB Hard Disk
* Sk. graf. Orchid Pro II 1024
* Monitor colore VGA 1024x768

Lire 3.960.000

"Stands out on the video benchmark tests"

PC MAGAZINE

Poly 486 - 33
* 128K Cache
* 4MB RAM
* 200MB Hard Disk
* Sk. graf. Orchid Pro II 1024
* Monitor colore VGA 1024x768

Lire 5.830.000

Tutti i sistemi sono forniti con licenza MS-DOS e manuali d'uso FDD da 3,5" o da 5,25", porte seriale e parallela.
Altre configurazioni sono disponibili su richiesta.
I prezzi sono IVA esclusa e possono subire variazioni senza preavviso.

Nuovi Annunci e Prodotti Speciali

- 386 il sistema può ospitare le CPU 386/25/33/40, 486sx, 486/25/33/50 consentendo un aggiornamento delle prestazioni a prezzi competitivi.
- Sceda Grafica VGA Edsun (Continue Edge Graphics) rende disponibile una risoluzione fino a 2500x2500 con 700.000 colori su monitor VGA standard.
- 486 a 50 Mhz con 256 Kb di memoria cache. "TOP GUN"
- Scheda Accelleratrice per PS/2 modelli 50/60s. Aumenta la velocità della CPU a 20 Mhz con 32 Kb di memoria cache.

Per informazioni e ordini telefonici chiamateci al numero:

(011) 4373313

Linea diretta assistenza tecnica
tel. (011) 489059

POLY SYSTEMS

Polywell Computers Inc.
61"C" airport Boulevard
South S.Francisco CA 94080 U.S.A.
FAX(415)583-1974

importatore e distributore esclusivo per Italia

M.C.E. Manutenzioni e Costruzioni Elettroniche srl
Via Capellina 12 - 10144 TORINO
FAX (011) 4730512

M.C.E. in Italia è:
- *Assistenza tecnica*
- *Reti locali*
- *Prodotti d'avanguardia*