Acta Universitaria
DIRECCIÓN DE APOYO A LA INVESTIGACIÓN Y AL POSGRADO

# An OOP Approach to Simplify MDI Application Development

*Sergio Ledesma*, Gustavo Cerda-Villafana*, Donato Hernández Fusilier* y Miguel Torres Cisneros*.*

**ABSTRACT**

The Multiple Document Interface (MDI) is a Microsoft Windows specification that allows managing multiple documents using a single graphic interface application. An MDI application allows opening several documents simultaneously. Only one document is active at a particular time. MDI applications can be deployed using Win32 or Microsoft Foundation Classes (MFC). Programs developed using Win32 are faster than those using MFC. However, Win32 applications are difficult to implement and prone to errors. It should be mentioned that, learning how to properly use MFC to deploy MDI applications is not simple, and performance is typically worse than that of Win32 applications. A method to simplify the development of MDI applications using Object-Oriented Programming (OOP) is proposed. Subsequently, it is shown that this method generates compact code that is easier to read and maintain than other methods (i.e., MFC). Finally, it is demonstrated that the proposed method allows the rapid development of MDI applications without sacrificing application performance.

**RESUMEN**

La Interfase para Múltiples Documentos (MDI) es una especificación del sistema operativo Microsoft Windows que permite manipular varios documentos usando un sólo programa. Un programa del tipo MDI permite abrir varios documentos simultáneamente. En un instante dado, sólo un documento es activo. Los programas del tipo MDI pueden desarrollarse usando Win32 o las clases fundamentales de Microsoft (MFC.) Los programas desarrollados usando Win32 son más rápidos que los programas que usan MFC. Sin embargo, éstos son difíciles de implementar promoviendo la existencia de errores. Cabe mencionar que el desarrollo de programas del tipo MDI usando MFC no es sencillo, y que su desempeño es típicamente peor que el de un programa del tipo Win32. Se propone un método que drásticamente simplifica el desarrollo de programas del tipo MDI por medio de la Programación Orientada a Objetos (POO.) Se demuestra que el método propuesto produce código que es más fácil de leer y mantener que el resultante por otros métodos (por ejemplo MFC). Adicionalmente, se demuestra que el método propuesto permite el rápido desarrollo de programas del tipo MDI sin afectar la velocidad del programa.

Recibido: 29 de Agosto de 2007
Aceptado: 29 de Octubre de 2008

Palabras clave:
Cliente; Múltiples Documentos; MDI; OOP; UML.

Keywords:
Client; Multiple Documents; MDI; OOP; UML.

## INTRODUCTION

In general, a window is divided in two sections, the client area and the non-client area as shown in Figure 1. The non-client area includes the title bar and the surrounding borders of the window. On the other hand, the client window covers completely the interior area delimited by the title bar and the window borders. A typical Windows application is responsible of drawing the client area, while the operating system is responsible of drawing and managing the non-client area.

Windows applications are made out of child windows. Each window is able to communicate with each other using the *Windows messages*. Con-

* División de Ingenierías del Campus Irapuato-Salamanca de la Universidad de Guanajuato. Correos electrónicos: selo@salamanca.ugto.mx, gcerdav@salamanca.ugto.mx, donato@salamanca.ugto.mx, mtorres@salamanca.ugto.mx.

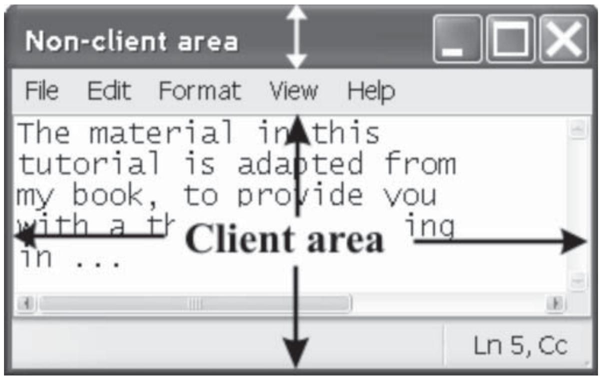


Figure 1. A Windows application showing the non-client area and the client area.

sequently, each window has a message queue to store its messages while it is busy and a function to process these messages. Plainly, a queue is a line that contains elements that are waiting for a service; i.e., when a customer goes to the bank, he enters a queue (line) and waits to be attended. On a Windows application, once a message is processed, it is removed from the queue and a new message may be processed. When the message queue is empty, the window object temporarily sleeps if no further processing is required. The function that processes the *Windows messages* is commonly known as the *window procedure*, and it explicitly determines the window behavior.

A Microsoft Windows class is a generic blueprint of a window object, and a Windows class needs to be registered before creating a window of that class. Specifically, the class registration process requires several parameters that establish window operation; the most important of these parameters is the *window procedure*. Typically, the *window procedure* must respond to the messages of interest for that window, and leave Windows to process the remaining messages through the use of the Application Program Interface (API) ::DefWindowProc. Note the use of the two colons before the function name to indicate that the function has been declared inside the global namespace. An application may register one or more Windows classes, or use a pre-registered Windows class. In general, Multiple Document Interface (MDI) applications have some similarities with typical Single document interface applications (SDI); however, there are some differences that will be addressed next, see the Microsoft Software Development Network (MSDN, 2005).

The main window of an MDI application, also known as the frame window, has a title bar, a menu, a sizing border, a toolbar, and three buttons to close, minimize and maximize the application as shown in Figure 2. The area inside the main window, also known as workspace, is the area where the document windows officially live. These documents windows are best known as MDI children, and can be moved inside the application workspace by the user. An MDI application clips its MDI child windows to its workspace preventing users from moving MDI child windows outside the frame window. MDI child windows do not have a toolbar or a menu; for this reason they only

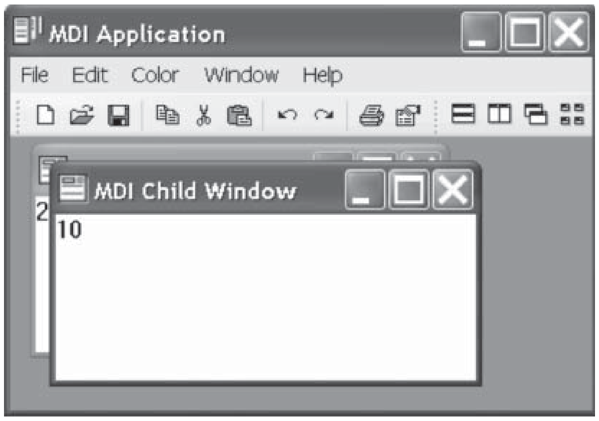


Figure 2. A typical MDI application showing two MDI child windows in cascade mode.

have a title bar as it can be seen from Figure 2.

The operating system, Microsoft Windows, allows several applications to share the mouse and the keyboard using the active-application concept. Thus, there is only one active application and this application receives mouse and keyboard input. Similarly, MDI extends this concept to document windows; as a result, only one MDI child window can be active inside an MDI application. The active child window has a highlighted title bar that allows the user to easily identify this window. Naturally, the active MDI child can be manipulated using the application menu and the application toolbar.

The menu and toolbar of an MDI application must be synchronized depending on the state of the active child and the application itself. When an MDI application has no children, its menu and toolbar may offer only a subset of viable options; usually the only available options are those of document creation (new empty document or new from the clipboard.) MDI applications may support different kinds of documents: i.e., a graph application may be able to create and manipulate pie charts as well as bar charts. For this type of applications, every time a document window becomes active an appropriate menu and toolbar is displayed at the top of the main window. Several commercial software products use MDI, i.e. Microsoft Excel and Corel Draw. Moreover, several researchers have been used MDI for simulation purposes, (see Bir *et al.*, 1992, Timmer *et al.*, 2000 and Kremer, 1993).

An MDI application has a special menu called the "Window" menu, which is located right before the "Help" menu as shown in Figure 2. The "Window" menu provides layout and activation options to operate the MDI children, and maintains a list of open documents for quick access and handling. Generally, this menu contains the items: Cascade, Tile Horizontally, Tile Vertically, and Close All; these options can also be accessed from the toolbar as shown in Figure 2 (see the icons that are right after the print and properties icons). In particular, the layout and activation options operate through a special window called the client window, which will be introduced next.

Inside the *frame window* there is a child window, called the *client window* that is responsible of controlling the position and size of the document windows. The *client window* appears invisible to most users because it fills the interior of the *frame window* and has a dark gray color. The pre-registered class MDICLIENT is used to create the *client window*, and its *window procedure* encapsulates most of the MDI required functionality. Because the *client window* is created using a pre-registered class, it is not necessary to provide a *window procedure* for this window. Even though the *frame window* receives the command messages through its menu, the *client window* and the active MDI child are responsible for processing most of these messages. Finally, it is important not to confuse the *client window* with the client area of a window; the *client window* applies only to MDI applications, while the client area is not a window but a specific area of it (see Newcomer, 1997 and Petzold, 1999).

In order to successfully create the main window, a typical Windows application starts by registering a Windows class. On the other hand, an MDI application typically starts by registering two Windows classes, one for the frame window and another one for the MDI children. Technically, this implies that a normal Windows application (non MDI) requires at least one window procedure, while an MDI application requires at least two.

As it was mentioned before, the *frame window* has a child, classically known as the *client window*. Thus, as soon as the *frame window* is processing the message WM_CREATE, it must proceed to create the *client window* by filling up a CLIENTCREATESTRUCT structure and call ::CreateWindow using the pre-registered class MDICLIENT. Right after these two windows have been successfully created, the MDI application is ready for user operation. Typically, the user will open an existing document or create a new one; in both cases the *frame window* will receive the message WM_COMMAND. Thus, the frame window must respond to this message by creating a new MDI child object. As it will be shown later, the proposed method reduces the application development time by hiding most of the MDI implementation details.

## Win32 Applications

Microsoft Corporation strategically develops their products using the Executive, which is a collection of APIs rarely known for most programmers. Because Microsoft does not publicly document the Executive, programmers must alternatively use an Executive subset called Win32 (see Williams, 2000). Because the Win32 APIs were deployed when computer memory and speed were limited, they are extremely efficient in nowadays computers. Unfortunately, the set of Win32 APIs was planned when most of the programming was done using plain C, and most of the application developers, back then, did not want to move to C++. Thus, Win32 has several shortcomings when OOP is required, and two of these are worth mentioning. First, window data storing is difficult to implement and requires either the use global variables or a custom data structure. Second, the *window procedure*, that is responsible of message processing, needs to be a global function or a static member function of a class; this means that the *window procedure* does not have any context information (at least not directly). Consequently, code written using Win32 contains a considerable number of global variables, and is difficult to read and maintain, not to be mentioned, prone to errors (Beveridge *et al.*, 1996).

## Microsoft Foundation Classes

Microsoft Corporation created a set of classes, better known as MFC, to simplify application development and tolerably provide OOP. To replace the *window procedure*, they introduced the concept of the *message map*, which is responsible for calling the appropriate function in response of a specific Windows message. Unfortunately, the *message map* unquestionably increases the message processing time, makes the object description confusing, and programmers eventually need to learn how to use it. Additionally, an application deployed using MFC has a direct overhead that does not exist in a Win32 application. On the bright side, Microsoft Visual Studio generously provides more than a few wizards to drastically simplify application development. There is a specific wizard to deploy MDI applications using MFC. This wizard allows the creation of complex projects to directly start working with. Once the wizard is completed, the programmer must regularly add and remove code; this can be simple if the programmer knows how to do it. However, adding or removing code with no previous MFC knowledge may break the application easily.

## PROPOSED METHOD

As it was mentioned previously, there are two major shortcomings of using Win32. We will describe, now, how they can be eliminated.

To simplify MDI application development, we propose the creation of a C++ class to represent a rectangular object that has a size, a position, and a *window procedure*. This *window procedure* may be deployed as a member function of a class, however, the APIs ::RegisterClass or ::RegisterClassEx accept only static functions for message processing. Consequently, a static *window procedure* will not be able to access any member variables or call any member functions that are not static. This issue can be solved using the API ::SetWindowLongPtr in combination with ::GetWindowLongPtr to store and retrieve context information (Yuan, 2000).

To simplify application development, it was suggested (see Petzold, 1999), to use a custom data structure to store window data, however, this traditional approach requires a lot of housekeeping and the resulting code is difficult to read and maintain. Alternatively, we propose to store the **this** pointer of a C++ class using the flag GWLP_USERDATA during the calls of ::SetWindowLongPtr and ::GetWindowLongPtr instead of storing a custom data structure. We will show that this approach results in clear and clean code and has the advantage that can be implemented on a base class making its use completely transparent to the programmer. This approach is motivated by the method proposed by Yuan, 2000.

To simplify the notation, assume that a data structure MsgInfo, as shown in the UML diagram of Figure 3, has been defined. This structure basically stores *Windows message* information, namely, a window handle and two context-parameters wParam and lParam of type WPARAM and LPARAM, respectively. After providing appropriate background information on MDI technology, we proceed to describe in detail the proposed method.

The UML diagram in Figure 4 describes the proposed classes: *Window, MdiChild* and *MdiFrame*. Observe that an appropriate namespace, called Win, has been defined to enable us to re-use the keywords already defined inside the global space and avoid name clashes. In this diagram, any computer screen object is represented by the base class *Window*; this class is abstract due to the virtual member function *Window::GetClassName* and the protected constructor of the class; thus, a *Window* object can be created only by class derivation (note that abstract classes and abstract methods are denoted in an UML diagram using italics).

As it was previously established, Microsoft Windows requires registering a Windows class before creating an object of that class. Therefore, the member function *Window::GetClassName* must be called once during class registration, and then repeatedly for each object that is created. The implementation of the member function *Window::GetClassName* is simple; it only returns the class name that is used for class registration. Specifically, a Windows class name is a text string that helps the operating system to identify this class. This can be verified using Spy++, the standard tool provided by Microsoft Visual Studio that uses the Windows class name to find and spy windows for debugging purposes. The member function *Window::GetClassName* is called by the member functions *MdiFrame*::RegisterClass and *MdiChild*::RegisterClass to register one MDI frame class and one MDI child class, respectively.

The implementation of the Win::*Window* class is straightforward; the public static variable *Window*::<u>hInstance</u> becomes handy to store the application instance and is used for window creation and resource loading (note that static variables and functions are represented as underlined text following the UML notation). The member functions Create, Destroy, Update, Show are just simple wrappers for the standard APIs ::CreateWindow, ::DestroyWindow, ::UpdateWindow and ::ShowWindow, respectively. It is important to note that ::CreateWindowEx can be used instead of the most traditional API ::CreateWindow; the only difference between these two is that ::CreateWindowEx supports additional window styles called extended window styles (note that Microsoft has followed the consistent

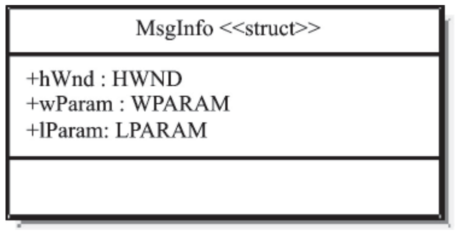


Figure 3. UML diagram of the MsgInfo structure.

notation of adding the letters Ex to the new versions of the traditional Windows APIs.)

One of the most interesting aspects of the Win::*Window* class is the implementation of the public operator HWND, which conveniently allows using a Win::*Window* object whenever a window handle is required. This can become pretty handy because several Windows APIs require a window handle. The implementation of this operator is just a return statement that provides the proper window handle that was stored during window creation, thus, there is no need to write wrappers for quite a lot of Windows APIs.

## The Win::*MdiFrame* Class

We propose the Win::*MdiFrame* class to ease the development of MDI applications. This class is depicted in Figure 4 and can be used to create the application window. Observe that the Win::*MdiFrame* class is abstract because it is directly derived from Win::*Window* and does not implement *GetClassName*. Additionally, Win::*MdiFrame* contains two abstract methods, *MdiFrame::OnCommand* and *MdiFrame::GetFirstChildID*; these can be implemented easily and they will be explicitly described on the RESULTS section. This class includes several helper functions that conveniently provide most of the functionality of a typical MDI application, for example, *MdiFrame*::SendMessageToActive allows sending a message to the active MDI child by calling internally *MdiFrame*::GetActiveWindow and then sending the respective message using the popular API ::SendMessage. Another handy function is *MdiFrame*::GetActiveWindow that allows getting the handle of the active child by sending a WM_MDIGETACTIVE to the client, and then validating the returned handle window. Despite the fact that most of the member functions of the Win::*MdiFrame* are pretty simple, they properly grant nearly all of the support required by an MDI application. The operation of the Win::*MdiFrame* class is described next.

The *MdiFrame*::CreateFrame function basically creates a local CREATESTRUCTURE variable and set the lpCreateParams value of this structure to the value of the **this** pointer of the current object, and internally calls ::CreateWindow (the function *Window*::Create can also be called). Remember that the standard API ::CreateWindow accepts a user define value than can be passed to the *window procedure* by using the parameter lpCreateParams. For this specific case, the **this** pointer must be sent as a user default value so that the new window object is able to store it through the use of the API ::SetWindowLongPtr. The step-by-step code to store and retrieve the **this** pointer of C++ class inside the window object will be discussed in detail now.

The function *MdiFrame*::RegisterClass registers the static member function *MdiFrame*::GWndProc for message processing. The letter G at the beginning of the function name denotes, in this case, "Generic". That is, Win::*MdiFrame* has a common window procedure that will be called for all the *MdiFrame* objects. The implementation of this function is shown in Figure 5. This function starts by checking if the message WM_NCCREATE has been receive; this message is received when the non-client area of the window is being created. This is supposed to be one of the first messages the *window procedure* receives. The documentation of WM_NCCREATE message specifically indicates that it is possible to transfer user defined information by using the parameter lpCreateParams of the CREATES-

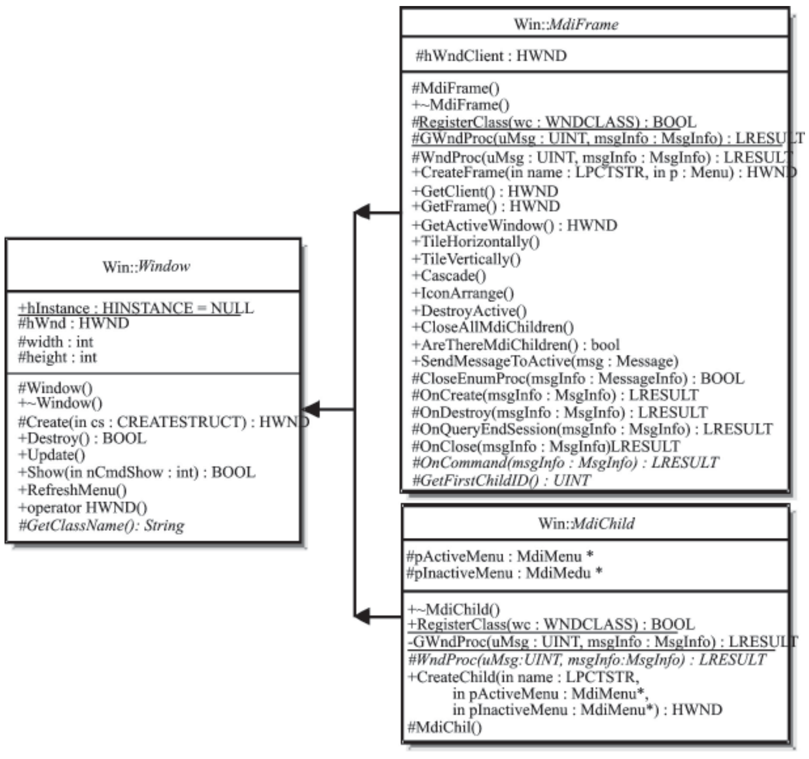
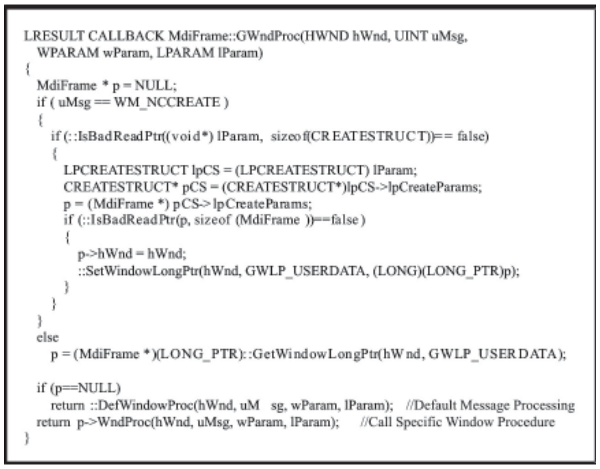

Win::MdiFrame
#hWndClient : HWND
#MdiFrame()
+~MdiFrame()
#RegisterClass(wc : WNDCLASS) : BOOL
#GWndProc(uMsg : UINT, msgInfo : MsgInfo) : LRESULT
#WndProc(uMsg : UINT, msgInfo : MsgInfo) : LRESULT
+CreateFrame(in name : LPCTSTR, in p : Menu) : HWND
+GetClient() : HWND
+GetFrame() : HWND
+GetActiveWindow() : HWND
+TileHorizontally()
+TileVertically()
+Cascade()
+IconArrange()
+DestroyActive()
+CloseAllMdiChildren()
+AreThereMdiChildren() : bool
+SendMessageToActive(msg : Message)
#CloseEnumProc(msgInfo : MessageInfo) : BOOL
#OnCreate(msgInfo : MsgInfo) : LRESULT
#OnDestroy(msgInfo : MsgInfo) : LRESULT
#OnQueryEndSession(msgInfo : MsgInfo) : LRESULT
#OnClose(msgInfo : MsgInfo)LRESULT
#OnCommand(msgInfo : MsgInfo) : LRESULT
#GetFirstChildID() : UINT
Win::Window
+hInstance : HINSTANCE = NULL
#hWnd : HWND
#width : int
#height : int
#Window()
+~Window()
#Create(in cs : CREATESTRUCT) : HWND
+Destroy() : BOOL
+Update()
+Show(in nCmdShow : int) : BOOL
+RefreshMenu()
+operator HWND()
#GetClassName(): String
Win::MdiChild
#pActiveMenu : MdiMenu *
#pInactiveMenu : MdiMedu *
+~MdiChild()
+RegisterClass(wc : WNDCLASS) : BOOL
-GWndProc(uMsg : UINT, msgInfo : MsgInfo) : LRESULT
#WndProc(uMsg:UINT, msgInfo:MsgInfo) : LRESULT
+CreateChild(in name : LPCTSTR,
in pActiveMenu : MdiMenu*,
in pInactiveMenu : MdiMenu*) : HWND
#MdiChil()

Figure 4. UML diagram showing the classes: *Window, MdiFrame* and *MdiChild*.

TRUCT variable that is passed during the initial call to ::CreateWindow or ::CreateWindowEx. Once the message WM_NCCREATE is being processed, the generic window procedure validates the parameter lParam to find out if it corresponds to the size of a variable of type CREATESTRUCT. If it is valid, the lParam is transformed by casting it to a CREATESTRUCT long pointer; peculiarly its lpCreateParams is another CREATESTRUCT structure. The lpCreateParams value of this last CREATESTRUCT variable is the user defined value, in this case, the **this** pointer of a C++ class. Once the pointer has been validated, the function proceeds to store it using the API ::SetWindowLongPtr using the flag GWLP_USERDATA as shown. Future calls of the generic *window procedure* will result in a call to ::GetWindowLongPtr to retrieve the original this pointer, and be able to call the specific *window procedure*. The proposed implementation completely hides the static nature of the generic window procedure, and programmers mind only on implementing the non-static function *MdiFrame*::WndProc. For this to work properly, it is very important not to forget setting the parameter lpCreateParams to the value of **this** pointer during the previous call to ::CreateWindow. In practice, this does not represent a problem because the function *MdiFrame*::CreateFrame does this automatically. Finally, it is imperative to mention that whenever an error occur during the execution of the generic *window procedure*, *MdiFrame*::GWndProc, the right thing to do is call ::DefWindowProc for default processing instead of just doing nothing.

The code shown in Figure 5 can be used on a debug version of the application; for a release version it is not mandatory to check the validity of the pointers received inside the CREATESTRUCT structure, and all the calls to ::IsBadReadPtr can be safely removed from the code, resulting in a very compact code.

Finally, we will address some of the actions that must occur during the execution of the virtual function *MdiFrame*::OnCreate, which is called during the processing of the message WM_CREATE. This function is declared as protected and is responsible of creating the *client window* using the pre-registered class MDICLIENT when calling the API ::CreateWindow. Once the *client window* has been successfully created, the function stores the *client window* handle in the variable **MdiFrame**::hWndClient. Because *MdiFrame*::OnCreate is declared as a virtual function, it is possible to overwrite this function to alternatively perform other initialization actions and then call the base class function; this can be useful for toolbar or rebar creation (a rebar is a Windows control that typically allows several toolbars, or other controls, to be positioned by the user).



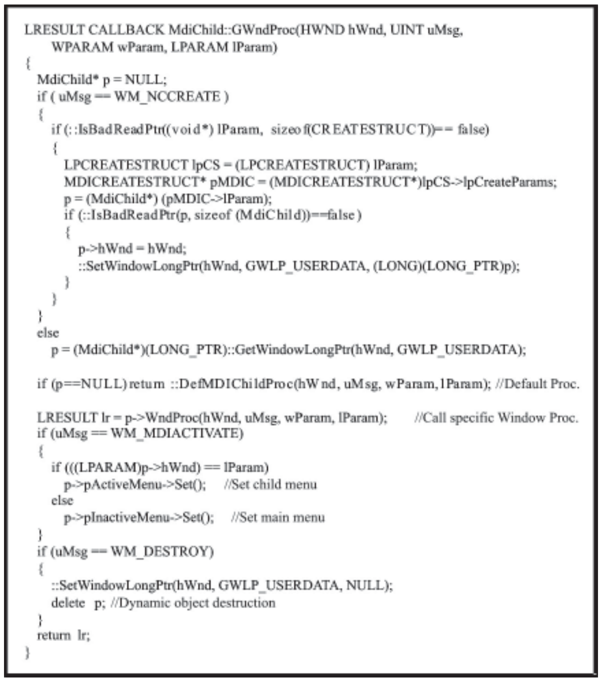
```
LRESULT CALLBACK MdiFrame::GWndProc(HWND hWnd, UINT uMsg,
  WPARAM wParam, LPARAM lParam)
{
  MdiFrame * p = NULL;
  if ( uMsg == WM_NCCREATE )
  {
    if (::IsBadReadPtr((void*) lParam,  sizeof(CREATESTRUCT))== false)
    {
      LPCREATESTRUCT lpCS = (LPCREATESTRUCT) lParam;
      CREATESTRUCT* pCS = (CREATESTRUCT*)lpCS->lpCreateParams;
      p = (MdiFrame *) pCS-> lpCreateParams;
      if (::IsBadReadPtr(p, sizeof (MdiFrame ))==false )
      {
        p->hWnd = hWnd;
        ::SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG)(LONG_PTR)p);
      }
    }
  }
  else
    p = (MdiFrame *)(LONG_PTR)::GetWindowLongPtr(hWnd, GWLP_USERDATA);

  if (p==NULL)
    return ::DefWindowProc(hWnd, uM   sg, wParam, lParam);   //Default Message Processing
  return  p->WndProc(hWnd, uMsg, wParam, lParam);     //Call Specific Window Procedure
}
```

Figure 5. Implementation of the generic *window procedure* for the Win::*MdiFrame* class.

## The Win::*MdiChild* Class

Figure 4 shows the UML diagram for the Win::*MdiChild* class. As it can be seen from this figure, this class is much simpler than the Win::*MdiFrame* class. There are, however, some special considerations that need to be taken due to the fact that objects of this class can be created dynamically; that is, a user can create as many MDI child windows as he wants at run time. To adequately provide dynamic object creation, we propose the function *MdiChild*::CreateChild as shown in Figure 4. *MdiChild*::CreateChild fills up a MDICREATESTRUCT to be able to send the message WM_MDICREATE to the client. The important thing to remember is to store the **this** pointer of the class using the lParam variable of the MDICREATESTRUCT so that we can successfully retrieve it and store it during the processing of the message WM_NCCREATE. Figure 6 shows the full code for the function *MdiChild*::GWndProc. This function is pretty similar to the function *MdiFrame*:GWndProc, however there are some evident differences. First, the lpCreateParams value is not a CREATESTRUCT but a MDICREASTRUCT. Second, instead of calling ::DefWindowProc for default processing, we must call ::DefMDIChildProc. Third, object destruction occurs dynamically during the processing of the message WM_DESTROY. It is important to mention that the main frame object must create a new *MdiChild* object using the operator **new**, and then call the function *MdiChild*::CreateChild. Finally, it can be seen from Figure 4 that the message WM_MDIACTIVATE plays an important role in MDI application development, and this message will be explained in detail next.



```
LRESULT CALLBACK MdiChild::GWndProc(HWND hWnd, UINT uMsg,
      WPARAM wParam, LPARAM lParam)
{
   MdiChild* p = NULL;
   if ( uMsg == WM_NCCREATE )
   {
      if (::IsBadReadPtr((void*) lParam,  sizeof(CREATESTRUCT))== false)
      {
         LPCREATESTRUCT lpCS = (LPCREATESTRUCT) lParam;
         MDICREATESTRUCT* pMDIC = (MDICREATESTRUCT*)lpCS->lpCreateParams;
         p = (MdiChild*) (pMDIC->lParam);
         if (::IsBadReadPtr(p, sizeof (MdiChild))==false )
         {
            p->hWnd = hWnd;
            ::SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG)(LONG_PTR)p);
         }
      }
   }
   else
      p = (MdiChild*)(LONG_PTR)::GetWindowLongPtr(hWnd, GWLP_USERDATA);

   if (p==NULL) return ::DefMDIChildProc(hWnd, uMsg, wParam,lParam); //Default Proc.

   LRESULT lr = p->WndProc(hWnd, uMsg, wParam, lParam);      //Call specific Window Proc.
   if (uMsg == WM_MDIACTIVATE)
   {
      if (((LPARAM)p->hWnd) == lParam)
         p->pActiveMenu->Set();    //Set child menu
      else
         p->pInactiveMenu->Set();    //Set main menu
   }
   if (uMsg == WM_DESTROY)
   {
      ::SetWindowLongPtr(hWnd, GWLP_USERDATA, NULL);
      delete   p; //Dynamic object destruction
   }
   return  lr;
}
```

Figure 6. Implementation of the static generic *window procedure* for the MdiChild class.

A user may have several documents open simultaneously, and each time a user clicks on an inactive window, the frame window sends a WM_MDIACTIVATE message to the client, which in turn, sends a MW_MDIACTIVATE message to both the window becoming active and the window becoming inactive. The active window receives this message as a request to become inactive, while the inactive window receives it as a notification that it is becoming active. An MDI child window may prevent itself of losing activation by processing the WM_NCACTIVE message as indicated in the Software Development Kit better known as the SDK (see MSDA, 2005). Because users typically switch among different open documents, the WM_MDIACTIVATE message is strongly related with menu and toolbar activation, as it will be explained.

Usually, an MDI application has at least two menus, one that is displayed before any document window has been created or opened, and another one that is displayed after the creation of the first MDI child. Both menus should be created right after the frame and child classes have been registered. One menu must be set as soon as the frame window is created, while the other one should be set once the client has at least one child. Because one of the menus is attached to the frame window when it is destroyed, it is not necessary to destroy the initial menu; however, the other menu needs to be destroyed explicitly.

Users expect the toolbar and the menu to display as enabled only those options that apply to the current state of the application. That is why the WM_MDIACTIVATE message is so important. Each time a window document receives the WM_MDIACTIVATE message the lParam parameter has the handle of the window that is becoming active. This is the perfect moment for menu and toolbar synchronization. For applications that handle more than one type of documents, this is the opportunity to switch to a different menu and/or toolbar. To set the menu, a MM_MDISETMENU must be sent to the client (note that the API ::SetMenu cannot be use for MDI applications as clearly indicated in the SDK.)

To manage menu synchronization, we expressly suggest the classes described in the UML diagram of Figure 7. At the top of the diagram, the base Win::Menu class is described. A Win::Menu object is a generic Windows menu that has an ID and some menu items that can appropriately be selected, enabled or

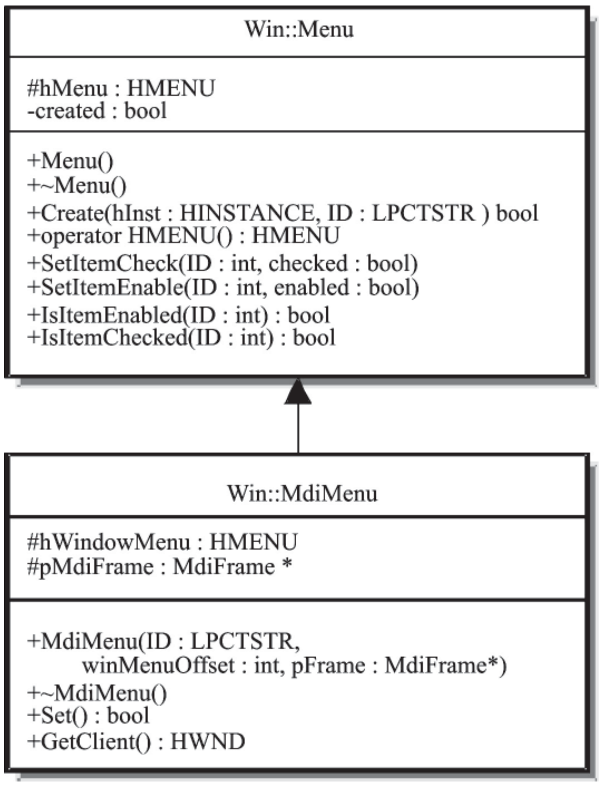

Win::Menu
#hMenu : HMENU
-created : bool
+Menu()
+~Menu()
+Create(hInst : HINSTANCE, ID : LPCTSTR ) bool
+operator HMENU() : HMENU
+SetItemCheck(ID : int, checked : bool)
+SetItemEnable(ID : int, enabled : bool)
+IsItemEnabled(ID : int) : bool
+IsItemChecked(ID : int) : bool
Win::MdiMenu
#hWindowMenu : HMENU
#pMdiFrame : MdiFrame *
+MdiMenu(ID : LPCTSTR,
winMenuOffset : int, pFrame : MdiFrame*)
+~MdiMenu()
+Set() : bool
+GetClient() : HWND

Figure 7. UML diagram showing the MdiMenu class.

checked. The Win::MdiMedu class, at the bottom of Figure 7, represents a menu for MDI applications. To create a Win::MdiMenu a Win::*MdiFrame* object is required as it can be seen from the constructor prototype. The second parameter of the constructor is an integer offset value to specify the position of the "Window" menu described previously, see Figure 2. The key function of this class is the MdiMenu::Set function, which sets (or activates) a menu. A typical MDI application creates two Win::MdiMenu objects (one when no MDI child exists and another when there is at least one MDI child). As it can be seen from Figure 6, menu activation is managed directly by the Win::*MdiChild*::GWndProc when the message WM_MDIACTIVATE is processed. As it can be seen from this figure, once the WM_MDIACTIVATE message is being processed, the function compares the current window handle with the value of the paramenter lParam, if they are equal the child menu is set using its *MdiChild*::Set function, otherwise the default menu is set.

## RESULTS

Once the Win::*MdiFrame* and Win::*MdiChild* classes have been clearly defined, developing an MDI application is straightforward. To illustrate this, consider Figure 8 and 9 that show the deployment of Multigraph, a graph editing applications using MDI technology. As it can be seen from Figure 8, the Multigraph class is derived directly from the abstract class Win::*MdiFrame*. The application must be able to create an object of type Multigraph, thus this class must be non-abstract and must implement *GetClassName*, *OnCommand* and *GetFirstChildID*. Implementing these functions is easy, *GetClassName* simply returns the text string "Multigraph", *GetFirstChildID* returns the ID of the first MDI child (i.e., 50 000), and *OnCommand* must respond to the application commands (a typical implementation might be a switch statement with case statements for each menu ID.)

Figure 9 shows the UML diagram for the Graph class. As it can be seen from this figure, the Graph class derives directly from the class Win::*MdiChild* which derives from Win::*Window*, therefore Graph must implement *GetClassName* and *MdiChild::WndProc*. Specifically, *GetClassName* returns the text string "Graph" and the function *MdiChild::WndProc* is called by the operating system for message processing. The Graph object must be able to store and draw a 2D graph on the surface of a window. The private variable Graph::**points** of type POINT[] is used for data storing while the function Graph::OnPaint is responsible for data drawing. The private function Graph::OnCommand is not called directly by the user; the client window redirects all WM_COMMAND messages to the active child. Graph::OnQueyEndSession and Graph::OnClose are not required, and can be implemented only on those applications that need to notify the user before closing the MDI child. The most important functions in the Graph class are Graph::OnMdiActivate and Graph::SyncMenuAndToolbar, their particular purpose is to synchronize the application menu and toolbar whenever the child becomes active. The implementation of Graph::OnMdiActivate is simple, it only calls the function Graph::SyncMenuAndToolbar, which, in turn, enables or disables the appropriate menu items by calling Win::Menu::SetItemEnable.

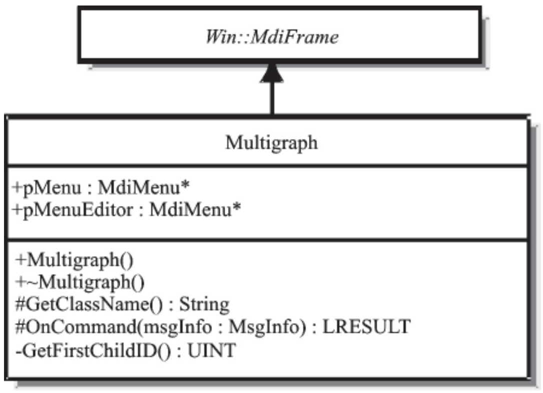


Figure 8. UML diagram showing the Multigraph MDI application.

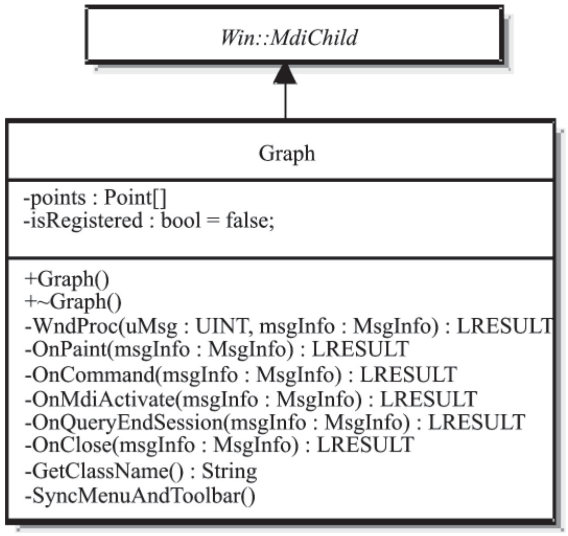

Win::MdiChild
Graph
-points : Point[]
-isRegistered : bool = false;
+Graph()
+~Graph()
-WndProc(uMsg : UINT, msgInfo : MsgInfo) : LRESULT
-OnPaint(msgInfo : MsgInfo) : LRESULT
-OnCommand(msgInfo : MsgInfo) : LRESULT
-OnMdiActivate(msgInfo : MsgInfo) : LRESULT
-OnQueryEndSession(msgInfo : MsgInfo) : LRESULT
-OnClose(msgInfo : MsgInfo) : LRESULT
-GetClassName() : String
-SyncMenuAndToolbar()

Figure 9. UML diagram showing the Graph class.

Note that the proposed method basically requires the implementation of two custom classes, while MFC requires the implementation of five classes. If each class is stored in a pair of files (a header file *.h and a source file *.cpp), the proposed method requires four files, in contrast, an MDI application deployed using MFC requires ten files, see Figure 12. On the other hand and despite the fact that a Win32 application may be implemented using only two files, its structure makes the code difficult to read and maintain, Thus, it can be seen that the proposed method is simpler than existing methods, and its structure makes the development, creation and maintenance of MDI applications easy.

In order to properly evaluate the proposed method, the Multigraph application was deployed using the proposed method and MFC. All the experiments were performed on a computer running Microsoft Windows XP on an Intel Pentium 4 CPU 3.2 GHz, 1.00 GB of RAM. First, we proceeded to measure the time the client requires to create a fixed number of MDI children by running the program 100 times and averaging the creation time of each experiment. The mean value obtained from these 100 measurements is shown in Figure 10. As it can be seen from this figure, MFC takes approximately twice the time to create the same amount of children than using the proposed method. This extra time may be due to the natural overhead of an MFC application.

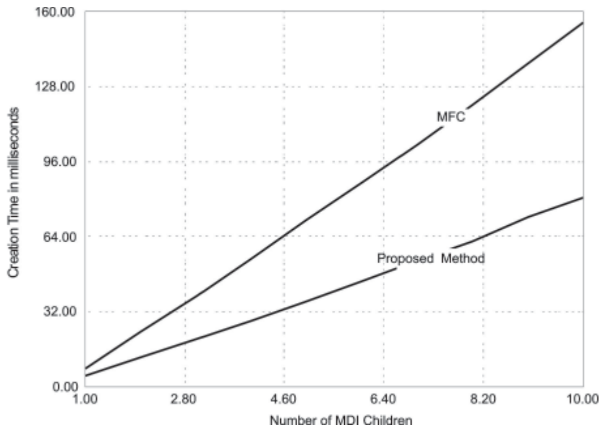


Figure 10. Creation Time for MDI children.

Another performance test that can be used to evaluate a Windows application is message-processing time. We proceeded to send a series of messages to the active child and measure the time it takes the active child to empty its queue. We performed these experiment 100 times. Figure 11 shows the mean value of these experiments. As it can be seen, the MFC application takes considerably longer to empty is message queue than the application deployed using the proposed method. This is a typical consequence of using the *message map* required by an MFC application.

## CONCLUSIONS

An MDI application is a special kind of application that allows managing several documents at the same time. MDI offers a common platform to deploy commercial application or perform research analysis in a shared environment. An MDI application can be deployed using MFC or Win32. While applications deployed using Win32 are much faster and efficient than applications deployed MFC, they usually contain several global variables and are prone to errors. On the other hand, Microsoft Visual Studio provides a set of wizards using MFC to simplify MDI application development. However, MFC adds an overhead that as a rule results in application performance degradation. We propose a method to simplify the development of MDI applications. We showed that our method allows creating clean code that is easy to read and maintain. The proposed method requires the derivation of only two custom classes, while MFC requires five. We also showed that our method offers a better performance than applications deployed using MFC; specifically the proposed method is generally twice faster than an MFC application.

Some years ago, two new technologies, Java and the .NET Platform, emerged changing the way we used to think about programming. These two technologies have been used successfully for application development. However, their main disadvantage is execution time;

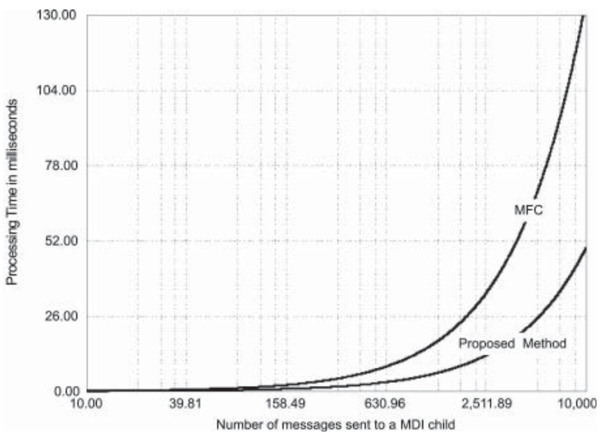


Figure 11. Message-processing time for MDI children.

even a MFC application is much faster than a Java or .NET application. Because the proposed method is based on pure Win32, it is several times faster than an application created with Java or .NET. On the other hand, Java code is more portable than a Win32 application; additionally, Java and the Platform .NET offer automatic object destruction which may improve code quality reducing *quality assurance* (QA) costs. Finding the correct balance between development time (which is directly related with development costs) and application performance depends usually on the type of application.

Last, it is important to mention that Win32 practically emerged with Windows 2000 using NT technology. That is, the Microsoft technology previously used only on the server versions of this operating system. Win32 is the native platform of Windows 2000, XP and Vista. Thus, Java, MFC and .NET applications require additional software installed to be executed by the computer.

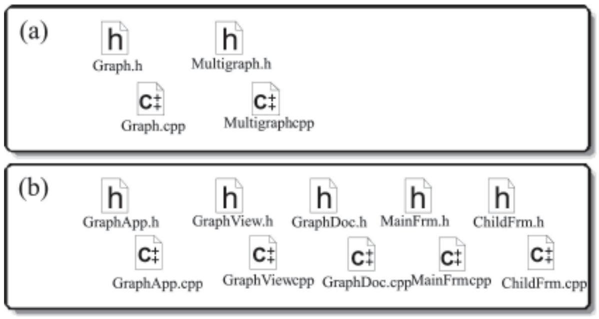


Figure 12. Typical MDI files using (a) the proposed method, (b) MFC.

## ACKNOWLEDGMENTS

The authors would like to thank PROMEP and CONACYT for their support.

## REFERENCES

Beveridge, J. and Wiener, R. (1996), *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley Professional. 251.

Bir M., Bodroghy E., Bor A., Knuth E. and Koves L. (1992) "The Design of DINE: A DIstributed NEgotiation Support Shell", *In: Decision Support systems: Experiences and Expectations*, Proc. of the IFIP TC8/WG8.3 Working Conference, Fontainebleau, North-Holland. 103 – 114.

Kremer, R. (1993), *A Concept Map Based Approach to the Shared Workspace*, Master Thesis, University of Calgary.

MSDN (2005). *The MSDN Library: An essential resource for developers using Microsoft tools, products and technologies*. (Win32 and COM Development > User Interface > Windows User Interface > Windowing) http://msdn.microsoft.com/library/

Newcomer J. M. (1997), *Win32 Programming, Addison-Wesley. Advanced Windows Series*. Chapter 17.

Petzold C. (1999). *Programming Windows* Fifth Edition, Microsoft Press, Redmond, Washington 98052. 1173.

Timmer J., Lauk M., Haussler S. and Radt. V. (2000), Cross-spectral analysis of tremor time series. *Int. J. Bifurcation Chaos*. V10-11. 2595 – 2610.

Williams A., (2000), *Windows 2000 Systems Programming Black Book*, Coriolis, Scottsdale, AZ 85260. 7 – 10.

Yuan F. (2000), *Windows Graphics Programming*, Prentice Hall Inc. Upper Saddle River, NJ 07458. 8 – 20.