How to design applications using ObjectWindows

Valery Sklyarov

Resumo **- Este artigo descreve as técnicas fundamentais aplicadas no desenvolvimento de aplicações com a biblioteca ObjectWindows da Borland International Inc. O artigo destaca, as ideias básicas introduzidas pelo sistema Windows, descreve algumas regras para a escrita de programas em C++ no ambiente ObjectWindows e apresenta uma aplicação orientada a objectos, em particular da linguagem C++, foram descritas em [1].**

Abstract **- This paper discusses the basic approaches involved in the use of Object Windows Library for application development. It emphasises the basic ideas introduced in Windows and ObjectWindows and recommends some rules for writing C++ object-oriented programs based on ObjectWindows supporting environment. When I say Windows I mean the operating system (subsystem). The paper has been prepared with a tutorial approach that makes the material accessible to students at an introductory level. I assume that readers know the fundamentals of C++ and Windows. The basic ideas of object-oriented programming techniques in general, and the C++ language in particular, were considered in [1]. The paper formally introduces the basic concepts of ObjectWindows developed by Borland International Inc. It does not assume prior knowledge of ObjectWindows.**

I. AN INTRODUCTION TO THE OBJECTWINDOWS LIBRARY

The Object Windows Library (**OWL**) contains a set of classes simplifying the creation and drawing of various components of a window on the screen. These components can be the following:

- rectangular windows themselves;
- dialog boxes (windows that support dialogue between a user and the computer);
- controls (buttons, schroll bars, etc.);
- menus (that enable you to specify some actions from a set of predefined actions);
- icons, gadgets and other graphical objects, etc.

ObjectWindows encapsulates a significant part of the Windows Application Programming Interface (**the Windows API**) allowing you to produce a Windows program very quickly and easily. If you want to draw the simplest default window on the screen you should perform the following two steps:

1. Define an object of the **OWL** class TApplication, for example:

```
# include <owl\applicat.h>   //including
      // the header file for the TApplication
      ..................................
            TApplication my_app;
```

2. Send a message "Draw the window". This is done by calling the function Run which is a member of the TApplication class, for example:

```
      my_app.Run();
```

As a result, the simplest ObjectWindows program will look something line the following:

```
      # include <owl\applicat.h>
      int OwlMain (int argc,char* argv[])
      {     TApplication my_app;
            return my_app.Run();
      }
```

Much in the way that every C/C++ language application has a single function *main*, every ObjectWindows application has a single OwlMain function. In other words you must use OwlMain instead of *main*. The OwlMain function returns an integer. That is why we used the statement:

```
      return my_app.Run();
```

The function Run also returns an integer.

II. HOW TO CONSTRUCT THE MAIN WINDOW ON THE SCREEN

When we are defining an object, we can make some initial settings by calling a special member function of a class that is called a **constructor**. A constructor is responsible for creating an object and initialising its data members. We could use a constructor to set the caption for the window in our example. Like many other ObjectWindows classes, TApplication has several constructors. The following constructor enables you to create an object (a class instance) from scratch:

```
      TApplication ( const char far* name=0 );
```

The constructor has just one parameter which has a default value of zero. We can call this constructor to

define an object. Since a constructor is a function, you can call it like the following:

```
return TApplication().Run();
```

Finally our simplest example can be presented as follows:

```
#include <owl\applicat.h>
int OwlMain(int, char**)
{  return TApplication("New").Run();  }
```

This program displays a rectangular window on the screen with the caption ‘New’. Let us look at the statement:

```
return  TApplication ( "New" ).Run();
```

The function Run() performs some predefined actions. Let us look at these actions in a little bit more detail. The call of the function Run in the statement above calls TApplication::InitApplication in the first instance, and TApplication::InitInstance for all subsequent instances. The expression TApplication::InitApplication says that the member-function InitApplication belongs to the TApplication class. If there is no error during the initialisation process, the function Run calls the function TApplication::MessageLoop.

The functions InitInstance and MessageLoop are very important, so let us look at them in more detail.

The first function calls TApplication::InitMainWindow, TWindow::Create and TWindow::Show.

The InitMainWindow builds a generic TFrameWindow object. The TFrameWindow class is responsible for controlling keyboard navigation, command processing for client windows, etc. Each window in an application has an associated TFrameWindow object, or an object of a class derived from TFrameWindow. You can override (redefine) the InitMainWindow function to build a customised main window object of TFrameWindow (or of a class derived from TFrameWindow). The functions TWindow::Create and TWindow::Show are used to create a window, and to display the window on the screen. If an error occurs, an exception is thrown. Both these functions belong to the TWindow class which enables windows to be created, and provides control of a window's behaviour supported by the **Windows API** routines.

The MessageLoop function runs throughout the lifetime of the application program. Let us briefly consider the main ideas of Windows which are fundamental to its architecture.

1. Windows and application programs communicate with each other through **messages**. The process of passing a message is achived by a function call.

2. If an application wants to obtain service from Windows it calls a **Windows API** function. This is sending a message to Windows through the API function name and its argument values.

3. Windows also sends messages to your application by calling a **window procedure**. If I say "Windows sends a message to the application", I mean that Windows calls a window procedure that belongs to the application, and passes a parameter to it which defines this message.

4. Messages can be either "**queued**" or "**non queued**". In the first case the messages are placed in an application's message queue by Windows, and then will be taken by the window procedure. In the second case the messages are sent to the application directly when Windows calls the window procedure. In both cases the application has one central point for message processing which is the window procedure.

5. The **OWL** TApplication::MessageLoop function is responsible for processing incoming messages from Windows. The function queries Windows for messages.

If a message is received, the function processes it by calling TApplication::ProcessAppMsg. Finally incoming messages will be retrieved from a message queue and dispatched to the window procedure supported by member functions of the TWindow class. For example, the TWindow::DefaultProcessing function installs a window procedure for the current application. The **OWL** window procedure is also used to respond to incoming messages. However the **OWL** introduces another mechanism which we will consider further.

If there are no messages in a queue, the function TApplication::MessageLoop calls TApplication::IdleAction. You can override (redefine) this function to perform background processing.

You can close an application by selecting the "Exit" or the "Close" item in the Windows menu or by pressing Alt-F4. Whenever you want to close an application, the main window's CanClose function will be called. You can override (redefine) this function to give the user a chance to perform some actions before closing (such as saving files for example).

III. BASIC STRUCTURES OF THE SIMPLEST APPLICATION PROGRAMS BASED ON OBJECTWINDOWS LIBRARY

If you want to override a function that belongs to the TApplication class, you need **to derive** your own class from TApplication. For instance, let us change the last program given in the first section.

```
#include <owl\applicat.h>
class my_app : public TApplication
{
public:
      my_app(const char far* name) :
         TApplication(name) {}
};
int OwlMain(int, char**)
{     return my_app("New").Run();   }
```

The statement:

```
class my_app : public TApplication
```

says that the user-defined class **my_app** is being derived from the **OWL** class TApplication. The constructor for the new class (its name is also my_app) simply calls the base class constructor, TApplication, and passes the parameter "name" on to this constructor. The window to be drawn will be the same as in the previous example.

In most circumstances, you need to override (redefine) the TApplication::InitMainWindow function for a useful application program. By default, InitMainWindow builds a frame window that is really useless because this window can not respond to any user input. Look at the following redefinition of the InitMainWindow function:

```
void my_app::InitMainWindow()
{  SetMainWindow(new TFrameWindow(0,"New"));  }
```

Normally, InitMainWindow performs the following steps:

1. Creates a TFrameWindow object (or an object of a class derived from TFrameWindow);
2. Calls the TApplication::SetMainWindow function that takes the pointer to the TFrameWindow object being created in point 1.

The function SetMainWindow returns a pointer to the old main window. In the case of a new application (the main window has not been set up yet), this function returns 0.

The TFrameWindow class has two constructors. Let us look at the constructor which enables you to create new frame window from scratch.

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             BOOL shrinkToClient = FALSE,
             TModule *module = 0);
```

where:

1. You specify the argument **parent as 0** if you are creating the main window for your application which has no parent window. If you are creating a child window, the **parent** argument is a pointer to the parent window object.
2. The argument **title** is a pointer to the string that will be displayed as your main window caption.
3. The argument **clientWnd** is a pointer to an object associated with your client window. This object provides all necessary service. If **clientWnd** is 0, that there is no client window for your application.
4. If you specify the argument **shrinkToClient** as TRUE, the frame window should shrink to fit the client window. If the **shrinkToClient** is FALSE, that it will not fit the frame to the client window.
5. The argument **module** is a pointer to the TModule object for the TFrameWindow constructor. TModule objects encapsulate the initialisation and closing functions of a Windows Dynamic Link Libraries (**DLL**).

In our previous example we called a TFrameWindow constructor as the follows:

```
TFrameWindow(0,"New");
```

The entire program will look something like the following:

```
#include <owl\applicat.h>
#include <owl\framewin.h> // header file for
                       // TFrameWindow class
class my_app : public TApplication
{     public:
       my_app() : public TApplication() {}
       virtual void InitMainWindow(); // we want
// to override default InitMainWindow function
};
void my_app :: InitMainWindow()
{  SetMainWindow(new TFrameWindow(0,"New")); }
int OwlMain(int, char**)
{      return my_app().Run();        }
```

IV. INTERFACE ELEMENTS AND INTERFACE OBJECTS

Interface elements provide user input for an application program. They are windows, dialog boxes, and controls such as text controls, buttons, etc. Instances (objects) which are used to create interface elements and to support their behaviour are called **interface objects**. Generally interface objects contain member functions that are used for creating, initialising, managing, and destroying their associated interface elements.

From a programmer's point of view, an interface object can be considered to be a logical window. On the other hand, associated with an interface element there is a physical window that is really displayed on the screen.

All **OWL** interface objects are derived from the TWindow class which provides generic low-level functionality for a large variety of objects.

Creating an interface object includes two basic steps:

1. Calling the interface object constructor which builds the interface object ant sets its attributes.
2. Creating the interface element by calling either the Create or the Execute member function belonging to the interface object.

After the two steps described above, Windows creates the window (the interface element) and in the process sends a WM_CREATE message. The **OWL** interface object intercepts the WM_CREATE message and calls its protected member function SetupWindow.

Finally you can consider the interface object constructor as a place where you can do some initialisation before the interface element is created. In this case **HWindow=NULL**, where data member of the interface object, **HWindow**, provides an association between interface object and interface element. You can consider the SetupWindow member function as a place where you can do some initialisation after the interface element has

been created. In this case **HWindow** is a handle for the interface element to be created.

V. PARENTS AND CHILDREN

The Windows desktop is a parent for all main windows. Each main window may have children. A child window is an interface element which is controlled (managed) by another parent interface element. The relationship between parents and children is supported through parent-child links .

You can construct a child window object in the body of the constructor of its parent window. Suppose we want to draw a button in the main window. In this case our program will look something like the following:

```
#include <owl\button.h>  // header file for
                         // TButton class
#include <owl\applicat.h>
#include <owl\framewin.h>
class DrawButton : public TWindow
{     public:
       DrawButton(TWindow* parent = 0); // this
           // is the constructor declaration
            // for DrawButton class
};
// see below the constructor definition for
// DrawButton class
DrawButton :: DrawButton(TWindow* parent) :
                          TWindow(parent,0,0)
{      new Tbutton(this,
       -1,"my_button",100,50,80,30);   }

class my_app : public TApplication
{     public:
       my_app() : TApplication() {}
       virtual void InitMainWindow();       };

void my_app :: InitMainWindow()
       {       SetMainWindow(new
        TFrameWindow(0,"New",new DrawButton)); }

int OwlMain(int, char**)
{   return my_app().Run();   }
```

This very simple program really leads us on to several specific features of **OWL** programming, so we will consider this program in more detail.

1. The first important part of our program is the following:

```
void my_app :: InitMainWindow()
{      SetMainWindow(new
       TFrameWindow(0,"New",new DrawButton)); }
```

where the third parameter of the TFrameWindow constructor is a pointer to an object associated with our client window (see the TFrameWindow class constructor in Section III). The C++ operator *new* allocates memory for the DrawButton object (constructs the DrawButton object) and returns a pointer to the DrawButton object that has been constructed. The DrawButton interface object is then responsible for our main window client area, and can be used to provide all necessary services.

2. Now let us look at the DrawButton class constructor

```
DrawButton :: DrawButton(TWindow* parent) :
               TWindow(parent,0,0)
{      new Tbutton(this,
       -1,"my_button",100,50,80,30);   }
```

2.1. It calls the TWindow base class constructor (TWindow(parent,0,0));

2.2. It creates a child interface element by calling the TButton class constructor (new TButton(this,-1,"my_button",100,50,80,30);). When you construct a child-window object in its parent window's constructor, the interface element for the child window will be automatically created. In our case we will see the button in our client area.

The TWindow class has two constructors. We used a constructor that enables us to create the corresponding interface object from scratch. This constructor is:

```
TWindow (TWindow* parent,
              const char far* title,
              TModule* module);
```

where:

1. You are setting **parent=0** if you are creating the main window which has no parent window. In our example this value was assigned by default: **DrawButton(TWindow* parent = 0)**. If you are creating a child window, the **parent** is a pointer to the parent window object.

2. The **title** is a pointer to the string that is displayed as your main window caption. In our example we want to display the caption "New" taken from the TFrameWindow object. That is why we set the second parameter to 0.

3. The argument **module** is a pointer to a TModule object. This parameter is used if we want to construct a TModule object which serves as the object-oriented interface for an ObjectWindows **DLL**. In our example this parameter was set to 0.

Now let us look at TButton class constructor mentioned above. It enables you to define a button interface object from scratch.

```
TButton(TWindow *parent,
          int Id,
          const char far *text,
          int X,
          int Y,
          int W,
          int H,
```

```
                BOOL isDefault = FALSE,
                TModule* module = 0);
```

where:

1. The **parent** argument is a pointer to a parent window. Note the use of the pointer *this* in our example to link the TButton child window with the DrawButton parent window.
2. The **Id** parameter is a unique identifier (**ID**) for the TButton instance (see the next section) sent by the button to its parent window. Passing a **-1** value means that we don't want to respond to this message.
3. The **text** is a pointer to the string that is displayed as your button's name.
4. The **X** (horizontal position), **Y** (vertical position), **W** (width), and **H** (height) parameters define the location and dimensions of the TButton interface element. In our example: X=100, Y=50, W=80, H=30.
5. The Boolean parameter **isDefault** specifies whether (TRUE) or not (FALSE) the button is the **default button** (pressing the Enter key on the keyboard is equivalent to clicking the default button). To emulate the keyboard interface for windows with controls mentioned above, you need to call the TFrameWindow::EnableKBHandler function before.
6. The **module** parameter was explained earlier.

This program is fairly useless. If you execute it in Windows environment, you will see the main window on the screen, but if you click the button, using the mouse, no actions will be performed. In a real application we want to respond to a button click. We can do so if we are able to do the following:

1. To generate a message after our button is clicked.
2. To respond to the message generated.

The following section will explain the main ideas of ObjectWindows related to mentioned above questions.

VI. GENERATING AND RESPONDING TO MESSAGES

ObjectWindows introduces **response tables** which are used to handle all events in an application program. Let us demonstrate the main ideas behind this method by inserting the simplest response table in our example from the previous section. The source code of the example needs to be modified as follows.

```
#include <owl\button.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#define BUTTON_ID 101
class DrawButton : public TWindow
{     public:
       DrawButton(TWindow* parent = 0);
       void HandleButtonMessage();  // response
                     // function declaration
       DECLARE_RESPONSE_TABLE(DrawButton);  };

DEFINE_RESPONSE_TABLE1(DrawButton,TWindow)
   EV_COMMAND(BUTTON_ID,HandleButtonMessage),
END_RESPONSE_TABLE;
DrawButton :: DrawButton(TWindow* parent)
{      Init(parent,0,0);
       new TButton(this,BUTTON_ID,"my_button",
                   100,50,80,30);    }

// see below the response function definition
void DrawButton :: HandleButtonMessage()
{      MessageBox("was pressed","Button");   }
class my_app : public TApplication
{     public:
       my_app() : TApplication() {}
       virtual void InitMainWindow();        };
void my_app :: InitMainWindow()
{      SetMainWindow(new
       TFrameWindow(0,"New",new DrawButton)); }
int OwlMain(int, char**)
{    return my_app().Run();    }
```

The new version of our program has the following additions:

1. Our TButton class object is constructed as follows:

```
new TButton(this,BUTTON_ID,"my_button",
            100,50,80,30);
```

Here **Id** is equal to **BUTTON_ID** which is a constant defined by **#define BUTTON_ID 101** (the value of the constant is 101). You can also replace the **#define** directive with the statement:

```
       const WORD BUTTON_ID = 101;
```

where WORD is equivalent to unsigned short. You can choose any constant you want from those that are not used by the **OWL**. If you click the button, this button sends the message named BUTTON_ID. Since BUTTON_ID = 101, the constant 101 is used by application program (by the window procedure) to identify this particular message.

2. Our window class DrawButton has been declared as follows:

```
class DrawButton : public TWindow
{     public:
       DrawButton(TWindow* parent = 0);
       void HandleButtonMessage();
       DECLARE_RESPONSE_TABLE(DrawButton);
};
```

The class declares the **DECLARE_RESPONSE_TABLE(DrawButton)** macro, taking one argument which is the name, DrawButton, of the class. This macro tells the compiler to build an empty message mapping table. Along with the **response table declaration**, we must include in our program a **response table definition**. This can appear anywhere outside the class declaration body. The macro name, **DEFINE_RESPONSE_TABLE1**, ends with the

digit **1** indicating that the class (DrawButton) has just one parent class (TWindow). If you want to consider a class with two parents classes, then you must use the **DEFINE_RESPONSE_TABLE2** macro (see below an example in the Section XI). Generally speaking, this macro is defined as **DEFINE_RESPONSE_TABLE#** and takes **#+1** arguments:

- the name of the class for which you are defining the response table;
- a sequence of # names for each intermediate base class.

In our case the definition is the following:

```
DEFINE_RESPONSE_TABLE1(DrawButton,TWindow)
   EV_COMMAND(BUTTON_ID,HandleButtonMessage),
END_RESPONSE_TABLE;
```

Where DrawButton is our class to be declared, and TWindow is its base class. The predefined name, **END_RESPONSE_TABLE**, ends the response table definition. Statements between the macros **DEFINE_RESPONSE_TABLE#** and **END_RESPONSE_TABLE** are the response table entries. In our example there is just one entry. You must always place a **comma** after each response table entry and a **semicolon** after the **END_RESPONSE_TABLE** macro.

There are two basic kinds of entries: **processing user-defined messages** and **processing predefined** (Windows) **messages**. In our example, we want to process our (user-defined) message that is generated after our button is clicked. The **EV_COMMAND** macro mentioned above takes two arguments, and establishes a connection between the first and the second arguments. The first argument is the name of an incoming message. The second argument is the name of a function which is to be used to respond to the incoming message. This function belongs to the same class (DrawButton). Finally when we click our button, the HandleButtonMessage member function will be called.

3. Let us now look at the HandleButtonMessage response function definition.

```
void DrawButton :: HandleButtonMessage()
{     MessageBox("was pressed","Button");   }
```

The MessageBox is a member function of the TWindow class that displays a message box on the screen.

We have looked at how we can handle the simplest user defined messages using the command message macro, EV_COMMAND. The basic syntax for this macro is:

```
EV_COMMAND(CMD, HandlerName)
```

The macro calls the member function HandlerName when the command message **CMD** is received. The prototype for the response function HandlerName is:

```
void HandlerName(void);
```

There are some extra types of command message macros. Some of them will be considered later. The following section is devoted to processing predefined (Windows) messages.

VII. HANDLING PREDEFINED MESSAGES

ObjectWindows has a set of **predefined macros** for all Windows messages which are automatically generated in various standard (predefined) situations, for instance:

- if you press keyboard key;
- if you click a mouse button;
- if you move the mouse;
- if you destroy a window; etc.

Let us start with an example which demonstrates how to respond to a left mouse button click, and a right mouse button click. The program code will be the following:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
class my_app : public TApplication //**
{                                  //**
public:                            //** This is
 my_app() : TApplication() {}   //** application
protected:                        //**      class
 virtual void InitMainWindow(void); //**
};                                  //**

class my_win : public TWindow     //##
{                                 //##
public:                           //##
     my_win(TWindow* parent = 0); //##   This is
protected:                        //##  window
 void EvLButtonDown(UINT,TPoint&);//##   class
 void EvRButtonDown(UINT,TPoint&);//##
 DECLARE_RESPONSE_TABLE(my_win);  //##
};                                //##
DEFINE_RESPONSE_TABLE1(my_win,TWindow)
     EV_WM_LBUTTONDOWN,
     EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_win :: EvLButtonDown(UINT,TPoint&)
{ MessageBox("You pressed the left button"); }
void my_win :: EvRButtonDown(UINT,TPoint&)
{ MessageBox("You pressed the right button"); }
void my_app :: InitMainWindow()
{    SetMainWindow(new TFrameWindow(0,"New",
                   new my_win));     }
int OwlMain(int char**)
{    return my_app().Run();  }
```

The possible sequence of various actions during program execution could be the following: clicking the left mouse button; activating the EV_WM_LBUTTONDOWN entry of our response table; calling the message response function EvLButtonDown; displaying a message box on the

screen. Nearly the same sequence of actions could be considered when you press the right mouse button.

ObjectWindows sets a correspondence between incoming messages, response table entries, and member functions that respond to the messages. To write the correct macro to be used as an entry in the response table, preface the Windows message name with **EV_**. To find the response function name, remove the **WM_** from the Windows message name, and convert the name to lowercase with capital letters at word boundaries. You can find the names of Windows messages in reference manuals.

The functions EvLButtonDown and EvRButtonDown mentioned above, have the following prototypes:

```
void EvLButtonDown(UINT modKey, TPoint& point);
void EvRButtonDown(UINT modKey, TPoint& point);
```

where:

1. The **modKey** parameter of each function corresponds to the key flags parameter.
2. The **point** parameter of each function is an object that keeps horizontal and vertical coordinates of your main window in which the left mouse button was pressed. You can have access to these coordinates as the following: **point.x**, **point.y**. For example, let us replace the EvLButtonDown function code in the previous program by the following code:

```
void my_win :: EvLButtonDown(UINT,
                             TPoint& point)
{     char str[30];
      wsprintf(str,"x = %d, y = %d",
                   point.x, point.y);
      MessageBox(str);
}
```

In this case, the current mouse coordinates will be displayed in your message box. The function wsprintf is almost the same as C/C++ function sprintf.

VIII. HOW TO CLOSE YOUR APPLICATION PROGRAM

TApplication and all Window classes have, or inherit, a CanClose member function. When you intend to close your application you select the "Exit" or "Close" item in the Windows menu, or press Alt-F4. Whenever you want to close an application, the main window's CanClose function will be called. You can override (redefine) this function to give the user a chance to perform some actions before closing (such as saving files for example). Let us consider an example.

```
#include <owl\applicat.h>
#include <owl\framewin.h>
class my_app : public TApplication
{   public:
      my_app() : TApplication() {}
    protected:
      virtual void InitMainWindow(void);    };
class my_win : public TWindow
{    public:
      my_win(TWindow* parent = 0) :
               TWindow(parent,0,0) {}
     protected:
      virtual BOOL CanClose();      };
BOOL my_win :: CanClose()
{ return MessageBox("Do you want to close?",
                    "?",MB_YESNO) == IDYES;   }
void my_app :: InitMainWindow()
{   SetMainWindow(new TFrameWindow(0,"New",
                              new my_win));   }

int OwlMain(int, char**)
{     return my_app().Run(); }
```

If you click the YES button in the message box, then you confirm the close operation and your application program really will be closed (the overridden CanClose function returns TRUE value which allows the application to close). If you click the NO button in the message box then you want to cancel the close operation. In this case the CanClose function returns FALSE value which cancels closing the application.

The CanClose function has the following prototype:

```
virtual BOOL CanClose(void);
```

The function returns TRUE if the associated interface element can be closed. For a parent window, this function calls the corresponding CanClose member functions of its children, and returns FALSE if any of the child CanClose functions return FALSE.

IX. WINDOW OBJECTS

The high level interface objects are called **window objects**. These objects are dealing with windows and their children. They provide all necessary services. There are several different types of window objects:

- frame windows;
- layout windows;
- decorated frame windows;
- Multiple Document Interface (MDI) windows;
- gadget windows.

Window objects represent interface elements. Each window object and its corresponding interface element has the same handle which is stored in the **HWindow** data member of the window object.

Setting up an interface element includes three basic steps:

- constructing a window object;
- setting attributes of a window interface element (size of window, etc.);
- creating a corresponding window interface element.

Step 1. ObjectWindows provides you with a set of classes that are only an abstraction. On the other hand, the object to be constructed is a tangible entity which exists in time and in space. The key part of constructing a window object is related to allocation of computer memory for the object. After that you can find its data and function members in memory, and you can use them. However at the outset, some of the object's members have undetermined states. For example **HWindow** is equal to NULL meaning that it points to nowhere.

Step 2. This step is related to setting specific values for members of the window object to be created that are undefined or have default values.

Step 3. When you have constructed a window object, you should tell Windows to build the associated interface element. You can do this by calling the Create member function that belongs to the object to be constructed. This function performs the following actions:

- builds the corresponding interface element;
- sets **HWindow** to the handle of the interface element;
- sets attributes for the actual state of the interface element;
- calls the object's SetupWindow member function.

The main window of your application is automatically created by TApplication::InitInstance. You don't need to call Create to build the main window.

Frame windows, that we have already considered previously, provide a service for a client window. They manage widely-used application elements such as menus and tool bars. A client window within a frame can be responsible for a single task. Many frame window attributes can be set after the object has been constructed. For instance you can attach a menu, or set a new icon for your application program.

Decorated frame windows inherit all the functionality of frame windows and layout windows . In addition, they provide:

- adding special controls called decorations to the frame of the window;
- automatic adjustment of the children to accommodate the placement of decorations.

Here is the single constructor for TDecoratedFrame:

```
TDecoratedFrame(TWindow *parent,
                const  char far *title,
                TWindow *clientWnd,
                BOOL trackMenuSelection = FALSE,
                TModule *module = 0);
```

There is only one parameter (**trackMenuSelection**) here that has not been considered yet. This parameter lets you specify whether menu commands should be tracked (TRUE value). When tracking is on, the window tries to pass a string to the window's status bar.

MDI windows support the Multiple Document Interface for managing multiple windows or views associated with a single application.

Gadget windows are used to hold a number of gadgets. **Gadgets** can be considered to be small software tools for various types of control. They look like icons and you can use them in nearly the same way as push buttons.

Generally speaking the main difference between various window objects is determined by sets of member functions that belong to the corresponding window class. All window classes are derived from TWindow class.

Let's consider some examples. The first example shows how to set attributes.

```
. . . . . .
class my_win : public TWindow
{ . . . . . .
protected:
       TStatic *my_static;
        . . . . . .
};
. . . . . .
my_win::my_win() : TWindow(0,0,0)
{ . . . . . .
       my_static=new Tstatic(this,-1,"text",
                              200,50,200,15,50);
       my_static->Attr.Style=
(my_static->Attr.Style & ~SS_LEFT) | SS_CENTER;
       . . . . . .
}
. . . . . .
```

In this example we declare a pointer **my_static** to the **OWL** class, TStatic, that supports working with static text. Then we create an object using the C++ operator, *new*. The expression:

```
       my_static->Attr.Style & ~SS_LEFT;
```

deletes **SS_LEFT** default style (the left-justified text). The expression:

```
       Attr.Style |= SS_CENTER;
```

adds the new style, **SS_CENTER**, that sets the text as centred. You can find various other attribute values in the Borland reference manuals. Consider another example:

```
       . . . . . .
void my_app::InitMainWindow(void)
{      SetMainWindow(new
            TFrameWindow(0,"text",new my_win)));
       nCmdShow = SW_SHOWMAXIMIZED;  }
```

Note the **nCmdShow** data member of the TApplication class. You can set this variable as soon as the Run function begins, up until the time you call TApplication::InitInstance. It denotes that you can set **nCmdShow** in the InitMainWindow function. As a result you can change how your main window is displayed. In our case the window will be maximised.

The last example demonstrates how to attach an accelerator table to your application (the accelerators will be considered more detailed in the Section XI).

```
. . . . . .
void my_app::InitMainWindow(void)
{      SetMainWindow(new
            TFrameWindow(0,"text",new my_win)));
       GetMainWindow()->Attr.AccelTable =
                                  "MY_TABLE"; }
```

Where the TApplication::GetMainWindow function returns a pointer to the application's main window.

X. SKELETON OF AN OBJECTWINDOWS APPLICATION PROGRAM

Let us try to consider a skeleton for an ObjectWindows application program which will be refined step by step later. The first version of this skeleton is shown in Figure 1. There are 6 essential fragments in Figure 1, marked with 1,...,6. Let us consider each separate fragment in a little more detail.

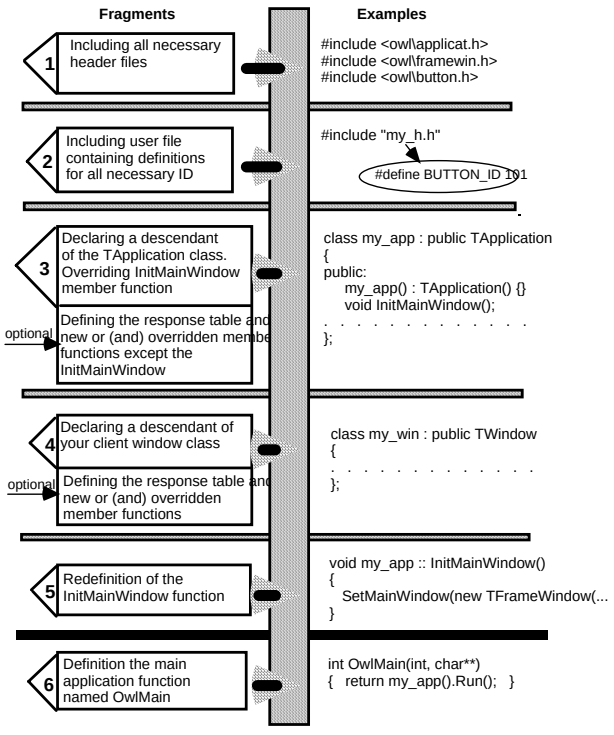

Fig. 1.

Fragment 1. Each class included into **OWL** has corresponding **header file**. If you use some classes in your application program, you must include all the header files for these classes in your application program.

Fragment 2. Most application programs interact with the user through various **window controls**. Each control has a **unique ID** associated with it. If you use a control for interaction, the **ID** (notification message) will be sent by this control to Windows, and subsequently to your application program (directly or vie a queue). An **ID** is really a **predefined constant**. Since you want to respond to control notification messages, you must define all necessary IDs in your program. It is worth-while to collect all these definitions in a single user-defined **.h** file (file with extension .h). Fragment 2 includes this file in your application program.

Fragment 3. The main task of an object that is derived from the TApplication class is to perform some default (predefined) actions to set up your main window. In most circumstances you need to override the TApplication::InitMainWindow function to customise your main window (add controls and decorations, change the default icon being used when your main window is minimised, and so on). In addition you can override some other functions that belong to the TApplication class. You can also consider the response table for this fragment.

Fragment 4. This fragment is used to declare a class which serves your client window. This class could be TWindow or its descendant, TDialog or its descendant, etc. The declaration of this class usually contains the following groups of data and function members:

1. Overridden version of the constructor. In the constructor body you can set some window interface element attributes, perform assignments for new data members, allocate memory for new data members, construct child windows, and so on.
2. Overridden version of the destructor that is usually used to deallocate memory that is allocated in the constructor.
3. New (usually protected) data members.
4. New or overridden (usually protected) member functions.
5. Overridden CanClose member function asking the user for confirmation that the main window should be closed.
6. Declaration of various message response functions.
7. Declaration of the response table.

Fragment 5. You override the InitMainWindow function to customise your main window. For these purposes you can add some statements in the function body that perform the following actions:

1. Attach a menu to your main window.
2. Construct a status bar.
3. Construct a control bar.
4. Insert the status bar and control bar into the frame (into your main window).
5. Change your main window attributes.
6. Set accelerators table.
7. Set a new icon that is used instead of predefined icon when the window is minimised.

Fragment 6. You define the OwlMain function instead of the *main* function in usual C/C++ programs. OwlMain has two arguments that are the same as the first two

arguments of the *main* function in C/C++ programs. You can use these arguments to pass the command line parameters onto OwlMain.

XI. HOW TO SOLVE VARIOUS PROGRAMMING TASKS

There are a number of basic things you can do with a class such as the following:

1. You can derive a new class from an existing class. You can refine the new class by extending its behaviour beyond that inherited from the base class. Let's consider an example:

```
// This is a place for fragments 1 and 3
class my_frame_win : public TFrameWindow
       // the beginning of fragment 4
{  public:
my_frame_win() : TFrameWindow(0,"Caption")  {}
   protected:
// you can declare new member functions for this
// class, response table, etc., for example:
       void EvLButtonDown(UINT, TPoint&);
       DECLARE_RESPONSE_TABLE(my_frame_win);  };
// you can define declared member function
// and a response table as shown below
DEFINE_RESPONSE_TABLE1(my_frame_win,
                          TFrameWindow)
   EV_WM_LBUTTONDOWN,
END_RESPONSE_TABLE;
void my_frame_win ::
       EvLButtonDown(UINT, TPoint& point)
{  MessageBox("You  pressed  the  left  button");}
       // the end of fragment 4
void my_app :: InitMainWindow(void)//fragment 5
{      SetMainWindow(new my_frame_win);      }
// This is a place for fragment 6
```

In this example, the TFrameWindow is an **OWL** class, and the **my_frame_win** is the new user-defined class which is derived from TFrameWindow.

2. You can define an object of a class. Class is an abstraction and represents a set of objects that share a common structure and common behaviour. On the other hand an object is a tangible entity, existing in time and in space. Our objects are created in computer memory (in space) and we can use them to solve a particular programming task (in time). Consider an example. Suppose we want to draw a rectangle in our client window and then automatically move it on the screen. ObjectWindows encapsulates the Windows Graphics Device Interface (**GDI**) which is a very powerful tool for working with graphics. **OWL TDC** class is the root class for encapsulating **GDI**. You can create a **TDC** object directly or you can use derived classes. For instance, the TClientDC class provides access to the client area that is owned by a window, and has the following constructor:

```
       TClientDC(HWND wnd);
```

where **wnd** is the handle of the owning window. As a result we can define an object **obj**, of the class TClientDC as follows:

```
       TClientDC obj(wnd);
```

Usually we are defining this object for our window class, in which case we can use the pointer *this* prefixed with an asterisk as the parameter for the constructor above. To use a class, you must create an instance of it. There are a number of ways to do this that are considered below.

1. You can use a standard declaration with arguments to be passed to the corresponding constructor, for example:

```
       TClientDC obj(*this);
```

If a class has a default constructor you can omit arguments, for instance:

```
       TMyApplication my_app;
```

2. You can define a pointer to an object, and then use the operator *new* to allocate space for an object, for example:

```
       TDC *dc;
       . . . . . .  .
       dc = new TClientDC(*this);
```

Since the **TDC** class is the root base class for other **GDI** classes (TClientDC included), we can use a pointer to the base class to allocate space for our object which is an instance of the TClientDC class. Let's look at the following example:

```
// Fragment 1
#define MY_ICON 100 // Fragment 2: the
              // definition of ID for our icon
// Fragment 3
class my_win : public Twindow // The
                    // beginning of fragment 4
{  public:
       my_win() : TWindow(0,0,0)  {}
   protected:
       TDC *dc;  // dc is a pointer to the TDC
                 // object
       int x,y;
       void SetupWindow();
       void EvLButtonDown(UINT, TPoint&);
       void EvTimer(UINT timerId);   // WM_TIMER
                     // is the Windows message
DECLARE_RESPONSE_TABLE(my_win);      };

DEFINE_RESPONSE_TABLE1(my_win, TWindow)
       EV_WM_LBUTTONDOWN,
       EV_WM_TIMER,
END_RESPONSE_TABLE;
```

```
void my_win :: EvLButtonDown(UINT,
                             TPoint& point)
{  x = point.x; y = point.y;
   dc = new TClientDC(*this); // Allocating
                      // memory for the object
   dc->Rectangle(x,y,x+100,y+100);
   delete dc;                          }

void my_win::SetupWindow()
{  TWindow::SetupWindow();
   SetTimer(1,1);  } // As fast as possible
void my_win::EvTimer(UINT)
{      dc = new TClientDC(*this);
       dc->Rectangle(x++,y++,x+100,y+100);
       if(x>500) x=y=0; if (y>800) x=y=0;
       delete dc; }   // The end of fragment 4
void my_app :: InitMainWindow(void)  // The
                    // beginning of fragment 5
{      SetMainWindow(new
          TFrameWindow(0,"my_win",new my_win));
       GetMainWindow()->SetIcon(this,MY_ICON);
}    // The end of fragment 5
// Fragment 6
```

In this example we derived the **my_win** class from the **OWL** class, TWindow. When you derive a new class, you can do three things:

1. Add new data members. In our example we added the dc pointer and two integers x and y.
2. Add new member functions. In our example we added two new member functions: EvLButtonDown and EvTimer.
3. Override inherited member functions. In our example we overrode the SetupWindow inherited member function.

We mentioned earlier (see the Section IV) that SetupWindow can be used to do some initialisation after the interface element has been created. In many cases you must first call the base class version of this function to make sure that the default process will be performed correctly. You can do this as follows:

```
TWindow::SetupWindow();
```

We override the SetupWindow member function to set up the timer. As a result the response table entry **EV_WM_TIMER** will be periodically activated (as fast as possible). Pressing the left mouse button forces a rectangle to be drawn in the window's client area. Then this rectangle will be periodically copied (see EvTimer function body). This will create the illusion that the rectangle is moving on the screen.

Now let's consider how to solve some of widely-used programming tasks. We will look only at the basic ideas that can be used in various ObjectWindows applications. The restricted volume of this paper does not allow these questions to be considered in more detail.

1. How to attach an icon to your program

Attaching an icon to a program was demonstrated in the previous example. We used the following statement:

```
GetMainWindow()->SetIcon(this,MY_ICON);
```

where TFrameWindow::SetIcon function sets the icon in the module passed as the first parameter, to the icon passed as a resource in the second parameter. The second parameter is the icon's **ID** that was defined in fragment 2 and is used to associate the icon resource in a ***.rc** script file with the application. This file will look something like the following:

```
#define MY_ICON  100
MY_ICON  ICON  "i1.ico"
```

where **i1.ico** is a file containing the icon's graphical image (bitmap). This file can be created using the **Resource Workshop** software (see below).

Suppose we want to move an icon on the screen In this case we can consider the following EvTimer function:

```
void my_win::EvTimer(UINT)
{  dc = new TClientDC(*this);
   ic = new
      TIcon(GetModule()->LoadIcon("NEW_ICON"));
   dc->DrawIcon(x++,y++,*ic);
   if(x>500) x=y=0; if (y>800) x=y=0;
   delete ic;
   delete dc;               }
```

We used the **OWL** TIcon class to construct our icon from an existing icon handle. This handle is returned by TModule::LoadIcon function. The pointer to an object of the TIcon class, **ic**, is declared as a protected data member of the **my_win** class:

```
TIcon *ic;
```

You must delete fragment 2 from the above program. In this case the resource script (***.rc**) file will look like the following:

```
NEW_ICON  ICON  "i1.ico"  // just one string
```

2. Resource Workshop overview

Resources are special components of Windows applications. Using resources you/ can change the text of messages, menus, icons, the cursor's shape and so on. The **Resource Workshop** enables you to create resources using visual programming techniques. As a result, you can draw the resources you need using the mouse and visual components displayed on the screen. The Resource

Workshop then creates the resource file, for example the script resource ***.rc** file. There are the following types of resources:

- **accelerators**, that are used as hot keys;
- **bit maps**, defining screen pictures;
- **cursors**, defining the shape of the cursor;
- **dialog boxes**, that are special windows containing controls;
- **fonts**, which can be created to support different alphabets;
- **icons** (small graphical images);
- **menus**, that are used to select an action from a set of predefined actions (items);
- **string tables**, that keep strings displayed, for example, as online help;
- user-defined resources.

3. How to use dialog boxes

Dialog boxes are windows that contain controls. We can consider a dialog box either as an entire window or as a child window. The first case is demonstrated in the following example.

```
// Fragment 1
#define VBX_CALL 100  // Fragment 2
// Fragment 3
class my_dlg : public Tdialog  // The
                   // beginning of fragment 4
{  public:
   my_dlg(TWindow* parent, TResID ResId):
  TDialog(parent, ResId), TWindow(parent) {}
 // You must call all base class constructors
 //  having non default arguments
   protected:
       void PressOK(); // The response function
       DECLARE_RESPONSE_TABLE(my_dlg);   };

DEFINE_RESPONSE_TABLE1(my_dlg,TDialog)
       EV_COMMAND(IDOK,PressOK),
END_RESPONSE_TABLE;

void my_dlg::PressOK()
{      Parent -> SendMessage(WM_CLOSE);      }
       // The end of fragment 4

void my_app::InitMainWindow()  //Fragment 5
{      SetMainWindow(new TFrameWindow(0,
           "Example",
       new my_dlg(0,VBX_CALL),TRUE)); }
// Fragment 6
```

The TFrameWindow class constructor sets up the **my_dlg** object as our client window (see Fragment 5). The **my_dlg** class declares a response table which is defined below. The table responds to **IDOK** notification message, and calls the **Press** message response function when you click the OK button in the dialog box. You can create the dialog box using the Resource Workshop. The function **Press** sends the message **WM_CLOSE** to the parent window. As a result the application will be closed. The following example demonstrates using a dialog box as a child window.

```
// Fragment 1
#define VBX_CALL 100  // Fragment 2
// Fragment 3
// The beginning of fragment 4
class my_dlg : public Tdialog // Declaring the
  // child window for user-defined my_win class
{  public:
       my_dlg(TWindow* parent, TResID ResId) :
           TDialog(parent, ResId),
           TWindow(parent) {}
protected:
       void EvRButtonDown(UINT, TPoint&);
       DECLARE_RESPONSE_TABLE(my_dlg);
};
// This table is used to respond to notification
// messages for my_dlg object
DEFINE_RESPONSE_TABLE1(my_dlg,TDialog)
       EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_dlg::EvRButtonDown(UINT, TPoint&)
{    MessageBox("my_dlg");   }

class my_win : public TWindow
{      public:
// Creating the dialog box as a child window
// dlg is a pointer to my_dlg object
       my_win() : TWindow(0,0,0)
         {   dlg = new my_dlg(this,VBX_CALL); }
protected:
       my_dlg *dlg;
       void EvRButtonDown(UINT, TPoint&);
       DECLARE_RESPONSE_TABLE(my_win);
};
// This table is used to respond to notification
// messages for my_win object
DEFINE_RESPONSE_TABLE1(my_win,TWindow)
       EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
void my_win::EvRButtonDown(UINT, TPoint&)
{ // Creation and execution of a dialog
  // box interface element
       dlg->Execute(); }
// The end of fragment 4
void my_app::InitMainWindow()  // Fragment 5
{      SetMainWindow(new TFrameWindow(0,
              "Example",new my_win)); }
// Fragment 6
```

After your dialog box has been created, you can use it to display various kinds of information and to input data.

Suppose you want to change the shape of the cursor for your dialog box. In this case you should add the following statement in your **my_dlg** class constructor:

```
{    SetCursor(GetModule(),C_ID);    }
```

where **C_ID** is an **ID** for resource defining the new shape. You can also use the GetApplication member function instead of GetModule. Both of them belong to the TWindow class. You can use almost the same technique to create customised cursors for different interface elements.

Windows and ObjectWindows also support predefined dialog boxes which provide a user interface for some **common tasks**, such as inputting data, and opening files, etc. These dialog boxes are called **common dialog boxes**.

4. How to attach a menu to you program

We do this in two steps. In the first step we create a menu using the Resource Workshop. In the second step we call the TFrameWindow::AssignMenu function in the InitMainWindow function body (fragment 5). Let's consider an example:

```
// Fragments 1 and 2
class my_dlg : public Tdialog // The beginning
                              // of fragment 4
{  public:
       my_dlg(TWindow* parent, TResID ResId):
              TDialog(parent, ResId),
              TWindow(parent)       {}
};            // The end of fragment 4
class my_app : public TApplication // The
                     // beginning of fragment 3
{ public:
       my_app() : TApplication() {}
  protected:
       void InitMainWindow();
       void call_dlg();
  DECLARE_RESPONSE_TABLE(my_app);    };

DEFINE_RESPONSE_TABLE(my_app,TApplication)
  EV_COMMAND(VBX, call_dlg),
END_RESPONSE_TABLE;
void my_app::call_dlg()
{  my_dlg(GetMainWindow(),VBX_CALL).Execute();  }
                     // The end of fragment 3
void my_app::InitMainWindow()              //
The beginning of fragment 5
{ SetMainWindow(new TFrameWindow(0, "Example"));
  GetMainWindow()->AssignMenu("COMMANDS");}
      // The end of fragment 5
// Fragment 6
```

The AssignMenu function sets the value of **Attr.Menu** to the supplied parameter which is the resource **ID** for the menu to be created. In our example, the menu is used to create and execute a dialog box interface element when you select the menu item associated with the VBX_CALL ID. The other item of the menu could be EXIT for instance.

5. How to decorate your window

If you want to decorate your window you should use the TDecoratedFrame class, **gadget windows** (with **gadgets**) and their member functions. Consider an example:

```
// Fragment 1, 2, 3 and 4
void my_app::InitMainWindow()  // The beginning
                               // of fragment 5
{  // Construct the decorated frame window
 TDecoratedFrame* frame = new TDecoratedFrame(0,
                    "title", new my_win, TRUE);
  // Construct a status bar

  TStatusBar *sb = new TStatusBar(frame,
                     TGadget::Recessed);
  // Construct a control bar
  TControlBar *cb = new TControlBar(frame);
  cb->Insert(*new TButtonGadget(CM_MINSK,
                                CM_MINSK);
  // Insert the status bar and control bar
  // into the frame
  frame->Insert(*sb, TDecoratedFrame::Bottom);
  frame->Insert(*cb, TDecoratedFrame::Top);
  // Set the main window and its menu
  SetMainWindow(frame);  }   // The end of
                             // fragment 5
// Fragment 6
```

The constructor for the TDecoratedFrame class was considered in the Section IX.

The TStatusBar class is an indirect descendant of the TGadgetWindow class. **Status bars** provide a few display options, and let you include multiple text strings which can be used, as online help, for example. These **string resources** can be created using the Resource Workshop. The following constructor:

```
TStatusBar *sb =
new TStatusBar(frame, TGadget::Recessed);
```

constructs the **status bar** for the parent window, accessed through the pointer **frame**. The value of **TGadget::Recessed** sets the border style.

The TControlBar class is an immediate descendant of the TGadgetWindow class. **Control bars** provide access for different **button gadgets**, which can be used, for example, to duplicate actions activated vie menu items. Each particular **button gadget** looks like a small icon that can be used in almost the same manner as a normal button. Gadgets can be created using the Resource Workshop. The following constructor:

```
TControlBar *cb = new TControlBar(frame);
```

constructs the **control bar** for the parent window accessed through the pointer, **frame**.

After you construct the status bar and the control bar, you must insert them in your window. For these purposes you can use the corresponding Insert member functions. The TGadgetWindow::Insert function inserts a gadget before or after a sibling gadget. In our case this function takes just one parameter which is an object of the TButtonGadget class, created by the C++ operator *new*. The TButtonGadget class constructor has two arguments. The first argument is an **ID** of a bitmap for the gadget created by the Resource Workshop. The second argument is the gadget's **ID**. The Insert function inserts the gadget after a sibling by default (the second argument of the Insert function can be assigned to either **After** or **Before** values). The TDecoratedFrame::Insert function inserts decorations (the status bar and the control bar in our case) above, below, to the left or to the right of the client window. In the program above, we inserted the status bar at the bottom (below), and the control bar at the top (above).

6. How to use VBX controls

Visual Basic (**VBX**) controls are supported through Windows **API functions**, and **OWL** classes. When you use the Resource Workshop you must first install **VBX** control libraries that are contained in either ***.VBX** or ***.DLL** files (use, for example, the menu option **Install control libraries** in the Resource Workshop **File** pop-up menu). Let's consider the following example:

```
// Fragment 1
#include <owl\vbxctl.h>  // Fragment 1
#include "vbxctlx.h"     // Fragment 2
              // The beginning of fragment 4
class my_dlg : public Tdialog,
               public TVbxEventHandler
{  public:
       my_dlg(TWindow* parent, TResID ResId):
              TDialog(parent, ResId),
              TWindow(parent){}
  protected:
       void EvMouseMove(VBXEVENT far* event);
       void EvClick(VBXEVENT far* event);
  DECLARE_RESPONSE_TABLE(my_dlg);
};

DEFINE_RESPONSE_TABLE2(my_dlg, Tdialog,
                            TVbxEventHandler)
  EV_VBXEVENTNAME(IDC_BIPICT1,"MouseMove",
                            EvMouseMove),
  EV_VBXEVENTNAME(IDC_BIPICT2,"Click",EvClick),
END_RESPONSE_TABLE;
void my_dlg::
          EvMouseMove(VBXEVENT far * /*event*/)
{  MessageBeep(0);    }

void my_dlg::EvClick(VBXEVENT far * event)
{      for(int i=0; i<100; i++)
       MessageBeep(0);        }
       // The end of fragment 4

class my_app : public TApplication  // The
                    // beginning of fragment 3
{ public:
       my_app() : TApplication() {}
  protected:
       void InitMainWindow();
       void call_dlg();
  DECLARE_RESPONSE_TABLE(my_app);    };

DEFINE_RESPONSE_TABLE(my_app,TApplication)
  EV_COMMAND(VBX, call_dlg),
END_RESPONSE_TABLE;

void my_app::call_dlg()
{  my_dlg(GetMainWindow(),VBX_CALL).Execute(); }
       // The end of fragment 3
void my_app::InitMainWindow() // Fragment 5
{  SetMainWindow(new TFrameWindow(0,"Example"));
   GetMainWindow()->AssignMenu("COMMANDS"); }

int OwlMain(int, char**)     // The beginning of
                             // fragment 6
{  TBIVbxLibrary vbxLib; // loading and
                 //  initialisation the library
      return my_app().Run(); }      // The end
                               // of fragment 6
```

If you want to handle events from a **VBX** control, you should derive your class from the TVbxEventHandler **OWL** class. The TVbxEventHandler needs to be mixed in with your windows class because it receives events from the **VBX** control which we want to intercept in our windows class. In our example we have declared our **my_dlg** class as follows:

```
class my_dlg : public TDialog,
               public TVbxEventHandler
```

Since our class **my_dlg** has two parents, we have defined below the **DEFINE_RESPONSE_TABLE2** macro (see also the Section VI). The macro **EV_VBXEVENTNAME** maps **VBX** events to handler functions. Let's consider an example:

```
EV_VBXEVENTNAME(IDC_BIPICT2,"Click",EvClick),
```

where the **IDC_BIPICT2** is the event **ID**, the "Click" is the event name (a predefined **VBX** event argument) and the EvClick is the response function. There are many predefined **VBX** event arguments such as "Click", "DblClick", "GotFocus", "KeyDown", "KeyPress", "KeyUp", "LostFocus", etc., that can be used in your

application programs. The MessageBeep is **Windows API** function which we call when we want to produce sound. You can define the **VBX** event **ID** as follows:

```
#define IDC_BIPICT1     101
#define IDC_BIPICT2     102
```

Suppose you have created two **VBX** controls (pictures) in your dialog box, using the Resource Workshop. Then when you create and show the dialog box interface element, you will see these pictures. If you move the cursor along the first **VBX** picture, you will hear sounds generated. If you click the second picture you will hear the long sound.

7 Transferring control data

Windows supports a data transfere between windows (dialog boxes), and a buffer which is usually a data member of the parent window. The buffer is composed of data fields for controls that are the check box, combo box, edit box, list box, radio button and scroll bar. It can be declared as follows:

```
struct my_buf {
       char fromEditBox[50];
       UINT RadioButton;     }
```

The basic rules for the buffer are:
- you should include only fields for controls whose data will be really transferred;
- declare fields in the same order that you define the controls in the corresponding constructor;
- if the transfer mechanism is enabled (use EnableTransfer and DisableTransfer member functions to enable and to disable this mechanism), the corresponding data is automatically transferred either when a window is created, or when a dialog box is created or executed.

You can define the transfer buffer as a public data member of your parent window class as follows:

```
my_buf transf_buf;
```

If the transfer buffer was declared in the **my_win** window class, you should include the following statement in the corresponding constructor:

```
      TransferBuffer =
 (void far*)&(((my_win *)Parent) -> transf_buf);
```

This statement assigns the address of the data buffer (**transf_buf**) to the predefined **TransferBuffer** pointer. It establishes the connection between the controls of the window (dialog box) and the buffer.

8. How to use MDI

The **MDI** makes it possible to open a number of child windows for different tasks, such as managing a database, or editing text. The **MDI** has the following components:
- the visible **MDI** frame window, which is an instance of the TMDIFrame class or its descendants;
- the invisible MDI client window, which is an instance of the TMDIClient class;
- the visible **MDI** child window, which can be dynamically created and removed. The child windows are instances of TMDIChild class or its descendant.

The sample structure of a program which involves the **MDI** mechanism is the following:

```
// Fragment 1, 2 and 3
class my_child : public TMDIChild   // The
                     // beginning of fragment 4
{  public:
      my_child(TMDIClient& parent);
. . . . . . . . . . . . . . . . . . . . };

my_child::my_child(TMDIClient& parent):
     TMDIChild(parent),TFrameWindow(&parent),
       TWindow(&parent)
{ . . . . . . . . . . . . . }
class my_client : public TMDIClient
{   public:
       my_client() : TMDIClient() {}
    protected:
       virtual TMDIChild* InitChild();      };
TMDIChild* my_client::InitChild(void)
{      TMDIChild* child;
 . . . . . . . . . . . . . . . . . .
       child = new my_child(*this);
       child -> SetIcon(GetModule(),ICON_ID);
       return child;                        }
       // The end of fragment 4
void my_app :: InitMainWindow(void)  // The
                    // beginning of fragment 5
{      MainWindow = new TMDIFrame("MDI",
              TResID(IDM_COMMANDS),
              *new my_client);             }
                     // The end of fragment 5
// Fragment 6
```

We have used the following TMDIFrame class constructor:

```
      TMDIFrame(const char far *title,
                     TresID menuResId,
                     TMDIClient &clientWnd =
                              *new TMDIClient,
                     TModule* module = 0);
```

It constructs an **MDI** frame window using the caption (**title**) and resource ID (**menuResId**). The third parameter specifies the client window.

9. How to create Windows help

Using Microsoft Windows you can create a **customised online help** system for your application program. Use the following basic steps for building a simple help file:

- prepare the topics for your help file and save them as Rich Text Format (**RTF**) files. You can use any available word processor that supports **RTF** format for this purpose, for example **Word for Windows**;
- prepare a contents topic and save it as **RTF** file;
- prepare a help project (***.HPJ**) file and save it as a text file;
- compile the topics into a help resource (***.HLP**) file.

You can then invoke the corresponding help topics from your application using predefined hot keys (to get extra information see **cwh.hlp** file containing 14 topics with explanations).

10. Tasks common to control objects

ObjectWindows introduces a number of controls, which are the following:

- push buttons (see Section V);
- check boxes that present two-states controls these are like flip-flops that are widely-used in various electronic devices;
- radio buttons which are used to select one of several mutually exclusive operations;
- group boxes that provide a means of grouping radio buttons or check boxes together;
- static text controls supporting static text, which can not be easily changed;
- edit text controls which support working with text that, unlike static text, can be readily changed;
- scroll bar controls that support tools that enable you to select a value within a predefined range of values;
- list box controls which support tools that enable you to select from a supplied list of items;
- combo box controls which support tools that enable you to combine an edit box with a list box.

The base class for controls is TControl, which is derived from TWindow. The following tasks are common to all control objects:

1. To construct the control object you can add a control object pointer data member to the parent window;
2. You can call the control object's constructor in its parent window's constructor;
3. You can change the **Attr.Style** data member inherited from TWindow, to set new control attributes using bitwise operations;
4. You can initialise a control in the SetupWindow member function (don't forget to call the base class SetupWindow member function first).
5. You can communicate with control objects by defining their **ID** which is one of the parameters for the object's constructor.

11. ObjectWindows technique common to various tasks

The following approaches are common to various programming tasks:

- use GetXXXXX member-functions to obtain something. For example, you can use the GetMainWindow function, returning a pointer to the application's main window, the GetClientWindow function, returning a pointer to the client window, etc. As a result you are able to access many member-functions that belong to different classes, vie pointers returning by GetXXXXX member-functions;
- use SetXXXXX member-functions to set something. For example, you can use SetIcon function to set the icon to the specified resource ID;
- use EvCommand member-functions to simplify the handling of **WM_COMMAND** notification messages when you want to select an item from a set of supplied items;
- use EnableXXXXX member-functions to enable something. For example, you can use the EnableKBHandler to enable keyboard navigation;
- use DisableXXXXX member-functions to disable something. For example, you can use the DisableTransfer function to disable the transfer mechanism;
- use SendXXXXX member-functions to send something. For example, you can use the SendNotification function to send a message from a child window to its parent;
- use LoadXXXXX member-functions to load something into memory. For example, you can use the LoadIcon function to load an icon resource into memory;
- the above approach can be also used for another common **OWL** member-function's names;
- when you override some member-functions, **you must call the original member-function first**. For example, you must do this when you redefine the EvLButtonDown function for the TEdit class. In this case the code looks something like the following:

```
void my_edit::EvLButtonDown(UINT modKeys,
                            TPoint& point)
{   TEdit::EvLButtonDown(modKeys, point);
  . . . . . . . . . . . . . . . . . . . }
```

XII CONCLUSIONS

We have discussed some common approaches to using ObjectWindows which provides very powerful tools for designing various kinds of application programs based on **object-oriented technology**. Briefly, the consequences resulting from what we have discussed are the following:

1. One of the most powerful tools for managing complexity is **hierarchical ordering**, which is organising related concepts into a tree structure with the most general concepts at the root. You can consider the ObjectWindows class hierarchy as such a tree structure.
2. When you are designing ObjectWindows applications you must focus on the following questions:

- **which OWL classes** you should use in your application program and how to use them;
- **how to build a class hierarchy** for your application program considering both single and multiple inheritance;
- **how to create objects** that are instances of given classes, and how to refine them for your application program;
- **how to decompose your program** into several simple parts which can be independently developed, and how to combine these parts later to produce your final (complex) program using parent-child links.

3. When you want to implement some ideas related to Point 2, you must be familiar with ObjectWindows. This is not easy because Windows and the OWL are really very complex software systems. The article briefly discusses some commonly-used approaches and methods which can be applied to solving a variety of application programs problems in the Windows environment. You can refer to [2,3,4,5] to obtain more information related to particular section of the article.

REFERENCES

[1] Valery Sklyarov From Procedural to Object-Oriented Programming. Electrónica e Telecomunicações, 1995, vol. 1, N 3, pp 217-223.

[2] Namir Clement Shammas What Every Borland C++ 4 Programmer Should Know. SAMS publishing, 1994, 898 p.

[3] Borland ObjectWindows for C++. Reference Guide, Borland International, Inc., 1993, 602 p.

[4] Borland ObjectWindows for C++. Programmer's Guide, Borland International, Inc., 1993, 418 p.

[5] Valery Sklyarov The Revolutionary Guide to Turbo C++. Birmingham, WROX, 1992, 352 p.