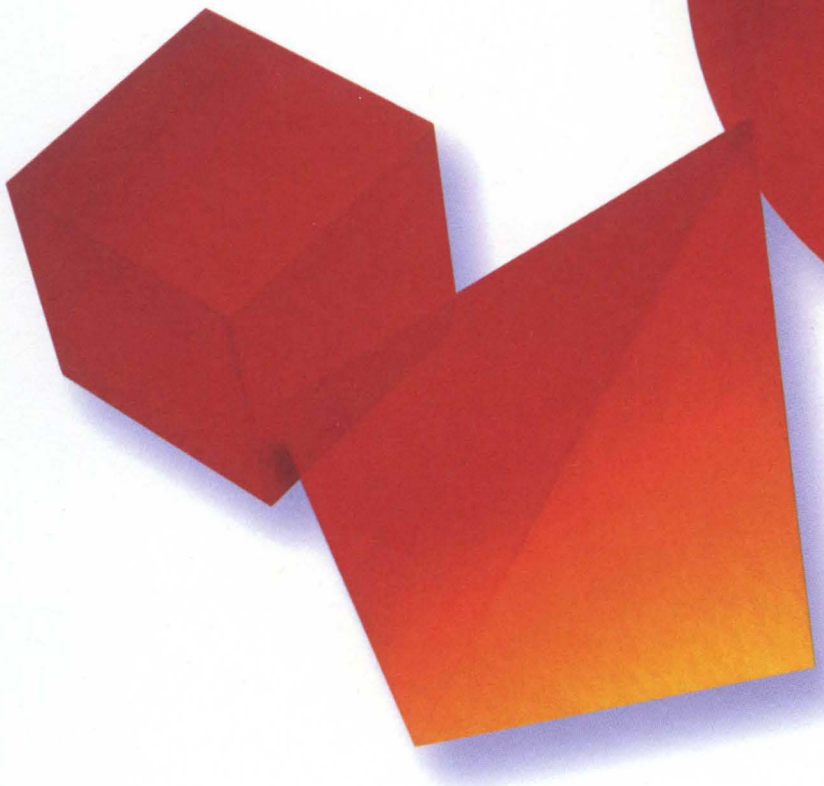
Tutorial

VERSION 2.5



Borland® ObjectWindows®

ObjectWindows Tutorial Borland

Tutorial

VERSION 2.5

Borland® ObjectWindows®

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R1094
9495969798-9 8 7 6 5 4 3 2
H1

Contents

Introduction

The ObjectWindows 2.5 tutorial teaches the fundamentals of programming for Windows using the ObjectWindows application framework. The tutorial is comprised of an application that is developed in twelve progressively more complicated steps. Each step up in the application represents a step up in the tutorial's lessons. After completing the tutorial, you'll have a full-featured Windows application, with items like menus, dialog boxes, graphical control bar, status bar, MDI windows, and more.

This tutorial assumes that you're familiar with C++ and have some prior Windows programming experience. Before beginning, it might be helpful to read Chapter 1 of the *ObjectWindows Programmer's Guide*, which presents a brief, nontechnical overview of the ObjectWindows 2.5 class hierarchy. This should help you become familiar with the principles behind the structure of the ObjectWindows class library.

For more detailed technical information on any subject discussed in this book, refer to the *ObjectWindows Programmer's Guide* and the *ObjectWindows Reference Guide*.

Getting started

Before you begin the tutorial, you should make a copy of the ObjectWindows tutorial files separate from the files in your compiler installation. Use the copied files when working on the tutorial steps. While working on the tutorial, you should try to make the changes in each step on your own. You can then compare the changes you make to the tutorial program.

Tutorial application

The tutorial application that you'll build when following the steps in this book is a line drawing application called Drawing Pad. While this application isn't very fancy, it does demonstrate many important ObjectWindows programming techniques that you'll use all the time in the course of your ObjectWindows development. Each step introduces a small increment in the application's features. You start with the most basic ObjectWindows application and, by the time you're finished with the last step, you'll have created a full-featured Windows application with a tool bar with bitmapped buttons on it, multiple document support, a status bar that displays menu and button hints, and even full OLE 2.0 server support.

Tutorial steps

Here's a summary of each step in the tutorial:

- In Step 1, you'll learn how to create the basic ObjectWindows application. This application has no real function except to show that an application *is* running.
- In Step 2, you'll learn how to use the ObjectWindows event-handling mechanism called response tables.
- In Step 3, you'll learn how to write text into a window by creating a device context object in the window and calling some of the device context object's member functions.
- In Step 4, you'll learn how to draw a line in a window using more functions of the device context object.
- In Step 5, you'll learn how change the size of the pen that you use to draw lines in the window. You'll also learn how to use a dialog box to get simple string input from the user.
- In Step 6, you'll learn how to take over the window's paint function, along with adding a menu to the window.
- In Step 7, you'll learn how to use some of the Windows common dialog boxes, specifically the File Open dialog box and File Save dialog box. You'll also learn how to check whether your application is ready to close when requested to do so by the user or the system, giving the application a chance to save files or clean up.
- In Step 8, you'll learn how to display and paint more than one line in the window using an array container to hold the information about all the lines in the drawing.
- In Step 9, you'll learn how to change the pen in the device context to let the user change the line color.
- In Step 10, you'll learn how to add decorations to the application, including a tool bar with bitmapped buttons on it and a status bar that displays hint text for menu items and tool bar buttons.
- In Step 11, you'll learn how to create a Multiple Document Interface (MDI) application, which lets the user of the application have a number of drawings open at once.
- In Step 12, you'll create a Doc/View application. Doc/View provides a programming model that lets you separate the object that actually contains your data (the document) from the object or objects that display your data on-screen (the views). This application is actually a Single Document Interface (SDI) application like Step 10.
- In Step 13, you'll combine the lessons of Step 11 and Step 12 to create an MDI Doc/View application.
- In Step 14, you'll learn to create an OLE 2.0 container from an MDI Doc/View application.
- In Step 15, you'll learn to create an OLE 2.0 server.

Files in the tutorial

The tutorial is composed of a number of different source files:

- Each step of the tutorial is contained in a file named STEPXX.CPP.
- Later steps in the application use multiple C++ source files. The other files are named STEPXXDV.CPP.
- A number of steps have a header file containing class definitions and the like. These header files are named STEPXXDV.H.
- A number of steps also have a corresponding resource script file named STEPXX.RC.

In each case, *XX* is a number from 01 to 15, indicating which step of the tutorial is in the source file.

Typefaces and icons used in this book

The following table shows the special typographic conventions used in this book.

Typeface	Meaning
Boldface	Boldface type indicates language keywords (such as **char**, **switch**, and **begin**) and command-line options (such as **–rn**).
Italics	Italic type indicates program variables and constants that appear in text. This typeface is also used to emphasize certain words, such as new terms.
`Monospace`	Monospace type represents text as it appears on-screen or in a program. It is also used for anything you must type literally (such as `TD32` to start up the 32-bit Turbo Debugger).
Menu | Command	This command sequence represents a choice from the menu bar followed by a menu choice. For example, the command "File | Open" represents the Open command on the File menu.

Note This icon indicates material you should take special notice of.

Chapter

1

Creating a basic application

To begin the tutorial, open the file STEP01.CPP, which shows an example of the most basic useful ObjectWindows application. Because of its brevity, the entire file is shown here: You can find the source for Step 1 in the file STEP01.CPP in the directory EXAMPLES\OWL\TUTORIAL.

```
//----------------------------------------------------------------------------
// ObjectWindows - (C) Copyright 1991, 1994 by Borland International
// Tutorial application -- step01.cpp
//----------------------------------------------------------------------------
#include <owl/applicat.h>
#include <owl/framewin.h>

class TDrawApp : public TApplication
{
  public:
    TDrawApp() : TApplication() {}

    void InitMainWindow()
    {
      SetMainWindow(new TFrameWindow(0, "Sample ObjectWindows Program"));
    }
};

int
OwlMain(int /* argc */, char* /* argv */ [])
{
  return TDrawApp().Run();
}
```

This simple application includes a number of important features:

- This source file includes two header files, owl\applicat.h and owl\framewin.h. These files are included because the application uses the *TApplication* and

TFrameWindow ObjectWindows classes. Whenever you use an ObjectWindows class you must include the proper header files so your code compiles properly.

- The class *TDrawApp* is derived from the ObjectWindows *TApplication* class. Every ObjectWindows application has a *TApplication* object—or more usually, a *TApplication*-derived object—generically known as the application object. If you try to use a *TApplication* object directly, you'll find that it's difficult to direct the program flow. Overriding *TApplication* gives you access to the workings of the application object and lets you override the necessary functions to make the application work the way you want.
- In addition to an application object, every ObjectWindows application has an *OwlMain* function. The application object is actually created in the *OwlMain* function with a simple declaration. *OwlMain* is the ObjectWindows equivalent of the *WinMain* function in a regular Windows application. You can use *OwlMain* to check command-line arguments, set up global data, and anything else you want taken care of before the application begins execution.
- To start execution of the application, call the application object's *Run* function. The *Run* function first calls the *InitApplication* function, but only if this instance of the application is the first instance (the default *TApplication::InitApplication* function does nothing). After the *InitApplication* function returns, *Run* calls the *InitInstance* function, which initializes each instance of an application. The default *TApplication::InitInstance* calls the function *InitMainWindow*, which initializes the application's main window, then creates and displays the main window.
- *TDrawApp* overrides the *InitMainWindow* function. You can use this function to design the main window however you want it. The *SetMainWindow* function sets the application's main window to a *TFrameWindow* or *TFrameWindow*-derived object passed to the function. In this case, simply create a new *TFrameWindow* with no parent (the first parameter of the *TFrameWindow* is a pointer to the window's parent) and the title "Sample ObjectWindows Program."

This basic application introduces two of the most important concepts in ObjectWindows programming. As simple as it seems, deriving a class from *TApplication* and overriding the *InitMainWindow* function gives you quite a bit of control over application execution. As you'll see in later steps, you can easily craft a large and complex application from this simple beginning.

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Application objects, along with their *Init** member functions, are discussed in Chapter 2 of the *ObjectWindows Programmer's Guide*.
- *OwlMain* is discussed in Chapter 2 of the *ObjectWindows Programmer's Guide*.
- *TFrameWindow* is discussed in Chapter 7 of the *ObjectWindows Programmer's Guide*.

Chapter

2

Handling events

You can find the source for Step 2 in the file STEP02.CPP in the directory EXAMPLES\OWL\TUTORIAL. Step 2 introduces response tables, another very important ObjectWindows feature. Response tables control event and message processing in ObjectWindows applications, dispatching events on to the proper event-handling functions. Step 2 also adds these functions.

Adding a window class

Add the response table to the application using a window class called *TDrawWindow*. *TDrawWindow* is derived from *TWindow*, and looks like this:

```
class TDrawWindow : public TWindow
{
  public:
    TDrawWindow(TWindow* parent = 0);

  protected:
    // override member function of TWindow
    bool CanClose();

    // message response functions
    void EvLButtonDown(uint, TPoint&);
    void EvRButtonDown(uint, TPoint&);

    DECLARE_RESPONSE_TABLE(TDrawWindow);
};
```

The constructor for this class is fairly simple. It takes a single parameter, a *TWindow* * that indicates the parent window of the object. The constructor definition looks like this:

```
TDrawWindow::TDrawWindow(TWindow *parent)
{
```

```
  Init(parent, 0, 0);
}
```

The *Init* function lets you initialize *TDrawWindow*'s base class. In this case, the call isn't very complicated. The only thing that might be required for your purposes is the window's parent, and, as you'll see, even that's taken care of for you.

Adding a response table

The only public member of the *TDrawWindow* class is its constructor. But if the other members are **protected**, how can you access them? The answer lies in the response table definition. Notice the last line of the *TDrawWindow* class definition. This declares the response table; that is, it informs your class that it has a response table, much like a function declaration informs the class that the function exists, but doesn't define the function's activity.

The response table definition sets up your class to handle Windows events and to pass each event on to the proper event-handling function. As a general rule, event-handling functions should be **protected**; this prevents classes and functions outside your own class from calling them. Here is the response table definition for *TDrawWindow*:

```
DEFINE_RESPONSE_TABLE1(TDrawWindow, TWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

You can put the response table anywhere in your source file.

For now, you can keep the response table fairly simple. Here's a description of each part of the table. A response table has four important parts:

- The response table declaration in the class declaration.
- The first line of a response table definition is always the DEFINE_RESPONSE_TABLE*X* macro. The value of *X* depends on your class' inheritance, and is based on the number of immediate base classes your class has. In this case, *TDrawWindow* has only one immediate base class, *TWindow*.
- The last line of a response table definition is always the END_RESPONSE_TABLE macro, which ends the event response table definition.
- Between the DEFINE_RESPONSE_TABLE*X* macro and the END_RESPONSE_TABLE macro are other macros that associate particular events with their handling functions.

The two macros in the middle of the response table, EV_WM_LBUTTONDOWN and EV_WM_RBUTTONDOWN, are response table macros for the standard Windows messages WM_LBUTTONDOWN and WM_RBUTTONDOWN. All standard Windows messages have ObjectWindows-defined response table macros. To find the name of a particular message's macro, preface the message name with EV_. For example, the macro that handles the WM_PAINT message is EV_WM_PAINT, and the macro that handles the WM_LBUTTONDOWN message is EV_WM_LBUTTONDOWN.

These predefined macros pass the message on to functions with predefined names. To determine the function name, substitute *Ev* for WM_, and convert the name to lowercase with capital letters at word boundaries. For example, the WM_PAINT message is passed to a function called *EvPaint*, and the WM_LBUTTONDOWN message is passed to a function called *EvLButtonDown*.

Event-handling functions

As you can see, two of the **protected** functions in *TDrawWindow* are *EvLButtonDown* and *EvRButtonDown*. Because of the macros in the response table, when *TDrawWindow* receives a WM_LBUTTONDOWN or WM_RBUTTONDOWN event, it passes it on to the appropriate function.

The functions that handle the WM_LBUTTONDOWN or WM_RBUTTONDOWN events are very simple. Each function pops up a message box telling you which button you've pressed. The code for these functions should look something like this:

```
void TDrawWindow::EvLButtonDown(uint, TPoint&)
{
  MessageBox("You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}

void TDrawWindow::EvRButtonDown(uint, TPoint&)
{
  MessageBox("You have pressed the right mouse button",
             "Message Dispatched", MB_OK);
}
```

This illustrates one of the best features of how ObjectWindows handles standard Windows events. The function that handles each event receives what might seem to be fairly arbitrary parameter types (all the macros and their corresponding functions are presented in Chapter 5 in the *ObjectWindows Reference Guide*). Actually, these parameter types correspond to the information encoded in the WPARAM and LPARAM variables normally passed along with an event. The event information is automatically "cracked" for you.

The advantages of this approach are two-fold:

- You no longer have to manually extract information from the WPARAM and LPARAM values.
- The predefined functions allow for compile-time type checking, and prevent hard-to-track errors that can be caused by confusing the values encoded in the WPARAM and LPARAM values.

For example, both WM_LBUTTONDOWN and WM_RBUTTONDOWN contain the same type of information in their WPARAM and LPARAM variables:

- WPARAM contains key flags, which specify whether the user has pressed one of a number of virtual keys.
- The low-order word of the LPARAM specifies the cursor's x-coordinate.

- The high-order word of LPARAM specifies the cursor's y-coordinate.

EvLButtonDown and *EvRButtonDown* also have similar signatures. The uint parameter of each function corresponds to the key flags parameter. The values that are normally encoded in the LPARAM are instead stored in a *TPoint* object.

Encapsulated API calls

You might notice that the calls to the *MessageBox* function look a little odd. The Windows API function *MessageBox* takes an HWND for its first parameter. But the *MessageBox* function called here is actually a member function of the *TWindow* class. There are a large number of functions like this: they have the same name as the Windows API function, but their signature is different. The most common differences are the elimination of handle parameters such as HWND and HINSTANCE, replacement of Windows data types with ObjectWindows data types, and so on. In this case, the window class supplies the HWND parameter for you.

Overriding the CanClose function

Another feature of the *TDrawWindow* class is the *CanClose* function. Before an application attempts to shut down a window, it calls the window's *CanClose* function. The window can then abort the shutdown by returning false, or let the shutdown proceed by returning true.

From the point of view of the application, this ensures that you don't shut down a window that is currently being used or that contains unstored data. From the window's point of view, this warns you when the application tries to shut down and provides you with an opportunity to make sure that everything has been cleaned up before closing.

Here is the *CanClose* function from the *TDrawWindow* class:

```
bool TDrawWindow::CanClose()
{
  return MessageBox("Do you want to save?", "Drawing has changed",
                    MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

For now, this function merely pops up a message box stating that the drawing has changed and asking if the user wants to save the drawing. Because there's no drawing to save, this message is fairly useless right now. But it'll become useful in Step 7, when you add the ability to save data to a file.

Using TDrawWindow as the main window

The last thing to do is to actually create an instance of this new *TDrawWindow* class. You might think you can do this by simply substituting *TDrawWindow* for *TFrameWindow* in the *SetMainWindow* call in the *InitMainWindow* function:

```
void InitMainWindow()
{
  SetMainWindow(new TDrawWindow);
}
```

This won't work, for a number of reasons, but primarily because *TDrawWindow* isn't based on *TFrameWindow*. For this code to compile correctly, you'd have to change *TDrawWindow* so that it's based on *TFrameWindow* instead of *TWindow*. Although this is fairly easy to do, it introduces functionality into the *TDrawWindow* class that isn't necessary. As you'll see in later steps, *TDrawWindow* has a unique purpose. Adding frame capability to *TDrawWindow* would reduce its flexibility.

The second approach is to use a *TDrawWindow* object as a client in a *TFrameWindow*. This is fairly easy to do: the third parameter of the *TFrameWindow* constructor that you're already using lets you specify a *TWindow* or *TWindow*-derived object as a client to the frame. The code would look something like this:

```
SetMainWindow(new TFrameWindow(0, "Sample ObjectWindows Program", new TDrawWindow));
```

With this approach, *TFrameWindow* administers the frame window, leaving *TDrawWindow* free to take care of its tasks. This makes for more discreet and modular object design. It also lets you easily change the type of frame window you use, as you'll see in Step 10.

Notice that the **new** *TDrawWindow* construction in the *TFrameWindow* constructor doesn't specify a parent for the *TDrawWindow* object. That's because there isn't yet anything to be a parent. The *TFrameWindow* object that will be the parent hasn't been constructed yet. *TFrameWindow* automatically sets the client window's parent to be the *TFrameWindow* once it has been constructed.

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Main windows are discussed in Chapter 2 of the *ObjectWindows Programmer's Guide*.
- Interface objects in general, such as windows, dialogs, controls, and so on, are discussed in Chapter 3 of the *ObjectWindows Programmer's Guide*.
- Response tables are discussed in Chapter 4 of the *ObjectWindows Programmer's Guide*.
- Window classes are discussed in Chapter 7 of the *ObjectWindows Programmer's Guide*.
- Predefined response table macros and their corresponding event-handling functions are listed in Chapter 3 of the *ObjectWindows Reference Guide*.

Chapter

3

Writing in the window

In Step 3, you'll begin working with the new window that was added to the application in Step 2. Instead of popping up a message box when the mouse buttons are pressed, the event-handling functions will get some real functionality—pressing the left mouse button will cause the coordinates of the point at which the button was clicked to be printed in the window, and pressing the right mouse button will cause the window to be cleared. You can find the source for Step 3 in the file STEP03.CPP in the directory EXAMPLES\OWL\TUTORIAL.

The code for this new functionality is in the *EvLButtonDown* function. The *TPoint* parameter that's passed to the *EvLButtonDown* contains the coordinates at which the mouse button was clicked. You'll need to add a **char** string to the function to hold the text representation of the point. You can then use the *wsprintf* function to format the string. Now you have to set up the window to print the string.

Constructing a device context

To perform any sort of graphical operation in Windows, you must have a device context for the window or area you want to work with. The same holds true in ObjectWindows. ObjectWindows provides a number of classes that make it easy to set up, use, and dispose of a device context. Because *TDrawWindow* works as a client in a frame window, you'll use the *TClientDC* class. *TClientDC* is a device context class that provides access to the client area owned by a window. Like all ObjectWindows device context classes, *TClientDC* is based on the *TDC* class, and is defined in the owl\dc.h header file.

TClientDC has a single constructor that takes an HWND as its only parameter. Because you want a device context for your *TDrawWindow* object, you need the handle for that window. As it happens, the *TWindow* base class provides an HWND conversion operator. This operator is called implicitly whenever you use the window object in places that require an HWND. So the constructor for your *TClientDC* object looks something like this:

```
TClientDC dc(*this);
```

Notice that the **this** pointer is dereferenced. The HWND conversion operator doesn't work with pointers to window objects.

Printing in the device context

Once the device context is set up, you have to actually print the string. The *TDC* class provides several versions of the *TextOut* function. Just like the *MessageBox* function in Step 2, the *TextOut* functions contained in the device context classes looks similar to the Windows API function *TextOut*. The first version of *TextOut* looks exactly the same as the Windows API version, except that the first HDC parameter is omitted:

```
virtual bool TextOut(int x, int y, const char far* str, int count=-1);
```

The HDC parameter is filled by the *TDC* object. The second version of *TextOut* omits the HDC parameter and combines the x- and y-coordinates into a single *TPoint* structure:

```
bool TextOut(const TPoint& p, const char far* str, int count=-1);
```

Because the coordinates are passed into the *EvLButtonDown* function in a *TPoint* object, you can use the second version of *TextOut* to print the coordinates in the window. Your completed *EvLButtonDown* function should look something like this:

```
void TDrawWindow::EvLButtonDown(uint, TPoint& point)
{
  char s[16];
  TClientDC dc(*this);

  wsprintf(s, "(%d,%d)", point.x, point.y);
  dc.TextOut(point, s, strlen(s));
}
```

You need to include the string.h header file to use the *strlen* function.

Clearing the window

TDrawWindow's base class, *TWindow*, provides three different invalidation functions. Two of these, *InvalidateRect* and *InvalidateRgn*, look and function much like their Windows API versions, but omitting the HWND parameters. The third function, *Invalidate*, invalidates the entire client area of the window. *Invalidate* takes a single parameter, a bool indicating whether the invalid area should be erased when it's updated. By default, this parameter is true.

Therefore, to erase the entire client area of *TDrawWindow*, you need only call *Invalidate*, either specifying true or nothing at all for its parameter. To clear the screen when the user presses the right mouse button, you must make this call in the *EvRButtonDown* function. The function would look something like this:

```
void TDrawWindow::EvRButtonDown(uint, TPoint&)
{
  Invalidate();
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Window classes are discussed in Chapter 7 of the *ObjectWindows Programmer's Guide*.
- Device contexts and the *TDC* classes are discussed in Chapter 14 of the *ObjectWindows Programmer's Guide*.

Chapter

4

Drawing in the window

You can find the source for Step 4 in the file STEP04.CPP in the directory EXAMPLES\OWL\TUTORIAL. In this step, you'll add the ability to draw a line in the window by pressing the left mouse button and dragging. To do this, you'll add a two new events, WM_MOUSEMOVE and WM_LBUTTONUP, to the *TDrawWindow* response table, along with functions to handle those events. You'll also add a *TClientDC* * to the class.

Adding new events

To let the user draw on the window, the application must handle a number of events:

- To start drawing the line, you have to look for the user to press the left mouse button. This is already taken care of by handling the WM_LBUTTONDOWN event.
- Once the user has pressed the left button down, you have to look for them to move the mouse. At this point, you're drawing the line. To know when the user is moving the mouse, catch the WM_MOUSEMOVE event.
- You then need to know when the user is finished drawing the line. The user is finished when the left mouse button is released. You can monitor for this by catching the WM_LBUTTONUP event.

You need to add two macros to the window class' response table, EV_WM_MOUSEMOVE and EV_WM_LBUTTONUP. The new response table should look something like this:

```
DEFINE_RESPONSE_TABLE1(TDrawWindow, TWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
END_RESPONSE_TABLE;
```

You also need to add the *EvLButtonUp* and *EvMouseMove* functions to the *TDrawWindow* class.

Adding a TClientDC pointer

The scheme used in Step 3 to draw a line isn't very robust:

- In Step 3, you created a *TClientDC* object in the *EvLButtonDown* function that was automatically destroyed when the function returned. But now you need a valid device context across three different functions, *EvLButtonDown*, *EvMouseMove*, and *EvLButtonUp*.
- You can catch the WM_MOUSEMOVE event and draw from the current point to the point passed into the *EvMouseMove* handling function. But WM_MOUSEMOVE events are sent out whenever the mouse is moved. You only want to draw a line when the mouse is moved with the left button pressed down.

You can take care of both of these problems rather easily by adding a new **protected** data member to *TDrawWindow*. This data member is a *TDC* * called *DragDC*. It works this way:

- When the left mouse button is pressed, the *EvLButtonDown* function is called. This function creates a new *TClientDC* and assigns it to *DragDC*. It then sets the current point in *DragDC* to the point at which the mouse was clicked. The code for this function should look something like this:

```
void
TDrawWindow::EvLButtonDown(uint, TPoint& point)
{
  Invalidate();
  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->MoveTo(point);
  }
   }
```

- When the left mouse button is released, the *EvLButtonUp* function is called. If *DragDC* is valid (that is, if it represents a valid device context), *EvLButtonUp* deletes it, setting it to 0. The code for this function should look something like this:

```
void
TDrawWindow::EvLButtonUp(uint, TPoint&)
{
  if (DragDC) {
    ReleaseCapture();
    delete DragDC;
    DragDC = 0;
  }
}
```

- When the mouse is moved, the *EvMouseMove* function is called. This function checks whether the left mouse button is pressed by checking *DragDC*. If *DragDC* is 0, either the mouse button has not been pressed at all or it has been pressed and released. Either way, the user is not drawing, and the function returns. If *DragDC* is valid, meaning that the left mouse button is currently pressed down, the function draws a line from the current point to the new point using the *TWindow::LineTo* function.

```
void
TDrawWindow::EvMouseMove(uint, TPoint& point)
{
  if (DragDC)
    DragDC->LineTo(point);
}
```

Initializing DragDC

You must make sure that *DragDC* is set to 0 when you construct the *TDrawWindow* object:

```
TDrawWindow::TDrawWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;
}
```

Cleaning up after DragDC

Because *DragDC* is a pointer to a *TClientDC* object, and not an actual *TClientDC* object, it isn't automatically destroyed when the *TDrawWindow* object is destroyed. You need to add a destructor to *TDrawWindow* to properly clean up. The only thing required is to call **delete** on *DragDC*. *TDrawWindow* should now look something like this:

```
class TDrawWindow : public TWindow
{
  public:
    TDrawWindow(TWindow *parent = 0);
~TDrawWindow() {delete DragDC;}

  protected:
    TDC *DragDC;

    // Override member function of TWindow
    bool CanClose();

    // Message response functions
    void EvLButtonDown(uint, TPoint&);
    void EvRButtonDown(uint, TPoint&);
    void EvMouseMove(uint, TPoint&);
    void EvLButtonUp(uint, TPoint&);

    DECLARE_RESPONSE_TABLE(TDrawWindow);
};
```

Note that, because the tutorial application has now become somewhat useful, the name of the main window has been changed from "Sample ObjectWindows Program" to "Drawing Pad":

```
SetMainWindow(new TFrameWindow(0, "Drawing Pad", new TDrawWindow));
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Event handling is discussed in Chapter 4 in the *ObjectWindows Programmer's Guide.*
- Device contexts and the *TDC* classes are discussed in Chapter 14 in the *ObjectWindows Programmer's Guide.*
- Predefined response table macros and their corresponding event-handling functions are listed in Chapter 3 in the *ObjectWindows Reference Guide.*

Chapter

5

Changing line thickness

You can find the source for Step 5 in the files STEP05.CPP and STEP05.RC in the directory EXAMPLES\OWL\TUTORIAL. In this step, you'll make the drawing capability in the application a little more robust. This step adds the ability to change the thickness of the line. To support this, you can add to the *TDrawWindow* class a *TPen* * drawing object and an **int** to hold the pen width.

Adding a pen

Add the pen to the window class by adding two **protected** members, *Pen* (a *TPen* *) and *PenSize* (an **int**). The most important changes that result from adding a pen to the window class are implemented in the *EvLButtonDown* and *EvRButtonDown* functions.

Initializing the pen

The *Pen* object and *PenSize* must be created and initialized before the user has an opportunity to draw with the pen. The best place to do this is in the constructor:

```
TDrawWindow::TDrawWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;

  PenSize = 1;
  Pen = new TPen(TColor::Black, PenSize);
}
```

The *TColor::Black* object in the *TPen* constructor is an **enum** defined in the owl\color.h header file. This makes the pen black. You'll learn more about this parameter of the *TPen* constructor later on in Step 9.

Selecting the pen into DragDC

To use the new pen object to draw a line, the pen has to be selected into the device context. The device-context classes have a function called *SelectObject*. This function is similar to the API function *SelectObject*, except that the ObjectWindows version doesn't require a handle to the device context.

You can use *SelectObject* to select a variety of objects into a device context, including brushes, fonts, palettes, and pens. You need to call *SelectObject* before you begin to draw. Add the call in the *EvLButtonDown* function immediately after you create the device context:

```
void
TDrawWindow::EvLButtonDown(uint, TPoint& point)
{
  Invalidate();

  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->SelectObject(*Pen);
    DragDC->MoveTo(point);
  }
}
```

Notice that ***Pen*** is dereferenced in the *SelectObject* call. This is because the *SelectObject* function takes a *TPen* **&** for its parameter, and *Pen* is a *TPen* *. Dereferencing the pointer makes *Pen* comply with *SelectObject*'s type requirements.

Changing the pen size

Having the ability to change the pen size in the application is of little use unless the user has access to that ability. To provide that access, you can change the meaning of pressing the right mouse button. Instead of clearing the screen, it now indicates that the user wants to change the width of the drawing pen. Therefore the process of changing the pen size goes into the *EvRButtonDown* function.

Once the user has indicated that he or she wants to change the pen width by pressing the right mouse button, you need to find some way to let the user enter the new pen width. For this, you can pop up a *TInputDialog*, in which the user can input the pen size.

Constructing an input dialog box

The *TInputDialog* constructor looks like this:

```
TInputDialog(TWindow* parent,
             const char far* title,
             const char far* prompt,
             char far* buffer,
             int bufferSize,
             TModule* module = 0);
```

where:

- *parent* is a pointer to the parent window of the dialog box. In this case, the parent is the *TDrawWindow* window. You can simply pass it in using the **this** pointer.
- *title* and *prompt* are the messages displayed to the user when the dialog box is opened. In this case, *title* (which is placed in the title bar of the dialog box) is "Line Thickness," and *prompt* (which is placed right above the input box) is "Input a new thickness:".
- *buffer* is a string. This string can be initialized before using the *TInputDialog*. If *buffer* contains a valid string, it is displayed in the *TInputDialog* as the default response. In this case, initialize *buffer* using the current pen size contained in *PenSize*.
- *bufferSize* is the size of *buffer* in bytes. The easiest way to do this is to use either a **#define** that is used to allocate storage for *buffer* or to use **sizeof**(*buffer*).
- *module* isn't used in this example.

To use *TInputDialog*, you must make sure its resources and resource identifiers are included in your source files and resource script files. These are contained in the file INCLUDE\OWL\INPUTDIA.RC. You should include INPUTDIA.RC in your resource script files and your C++ source files.

Executing an input dialog box

Once you've constructed a *TInputDialog* object, you can either call the *TDialog::Execute* function to execute the dialog box modally or the *TDialog::Create* function to execute the dialog box modelessly. Because there's no need to execute the dialog box modelessly, you can use the *Execute* function.

The *Execute* function for *TInputDialog* can return two important values, IDOK and IDCANCEL. The value that is returned depends on which button the user presses. If the user presses the OK button, *Execute* returns IDOK. If the user presses the Cancel button, *Execute* returns IDCANCEL. So when you execute the input dialog box, you need to make sure that the return value is IDOK before changing the pen size. If it's not, then leave the pen size the same as it is.

If the call to *Execute* does return IDOK, the new value for *PenSize* is in the string passed in for the dialog's buffer. Before this can be used as a pen size, it must be converted to an **int**. Then you should make sure that the value you get from the buffer is a valid pen width. Finally, once you're sure that the input from the user is acceptable, you can change the pen size. *TDrawWindow* now has a function called *SetPenSize* that you can use to change the pen size. The reason for doing it this way, instead of directly modifying the pen, is explained in the next section.

The *EvRButtonDown* function should now look something like this:

```
void
TDrawWindow::EvRButtonDown(uint, TPoint&)
{
  char inputText[6];

  wsprintf(inputText, "%d", PenSize);
```

```
  if ((TInputDialog(this, "Line Thickness",
                    "Input a new thickness:",
                    inputText,
                    sizeof(inputText))).Execute() == IDOK) {
    int  newPenSize = atoi(inputText);

    if (newPenSize < 0)
      newPenSize = 1;

    SetPenSize(newPenSize);
  }
}
```

Calling SetPenSize

To change the pen size, use the *SetPenSize* function. Although the *EvRButtonDown* function is a member of *TDrawWindow*, and as such has full access to the **protected** data members *Pen* and *PenSize*, it is better to establish a public access function to make the actual changes to the data. This becomes more important later, when the pen is modified more often.

For *TDrawWindow*, you have the **public** *SetPenSize* function. The *SetPenSize* function takes one parameter, an **int** that contains the new width for the pen. After opening the input dialog box, processing the input, and checking the validity of the result, all you need to do is call *SetPenSize*.

SetPenSize is a fairly simple function. To resize the pen, you must first delete the existing pen object. Then set *PenSize* to the new size. Finally construct a new pen object with the new pen size. The function should look something like this:

```
void
TDrawWindow::SetPenSize(int newSize)
{
  delete Pen;
  PenSize = newSize;
  Pen = new TPen(TColor(0,0,0), PenSize);
}
```

Cleaning up after Pen

Because *Pen* is a pointer to a *TPen* object, and not an actual *TPen* object, it isn't automatically destroyed when the *TDrawWindow* object is destroyed. You need to explicitly destroy *Pen* in the *TDrawWindow* destructor to properly clean up. The only thing required is to call **delete** on *Pen*. *TDrawWindow* should now look something like this:

```
class TDrawWindow : public TWindow
{
  public:
    TDrawWindow(TWindow *parent = 0);
```

```
    ~TDrawWindow() {delete DragDC; delete Pen;}

    void SetPenSize(int newSize);

  protected:
    TDC *DragDC;
    int PenSize;
    TPen *Pen;

    // Override member function of TWindow
    bool CanClose();

    // Message response functions
    void EvLButtonDown(uint, TPoint&);
    void EvRButtonDown(uint, TPoint&);
    void EvMouseMove(uint, TPoint&);
    void EvLButtonUp(uint, TPoint&);

    DECLARE_RESPONSE_TABLE(TDrawWindow);
};
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- The *TInputDialog* class and dialogs in general are discussed in Chapter 9 in the *ObjectWindows Programmer's Guide.*
- Device contexts and the *TDC* classes are discussed in Chapter 14 in the *ObjectWindows Programmer's Guide.*
- The *TPen* class is also discussed in Chapter 14 in the *ObjectWindows Programmer's Guide.*

Chapter

6

Painting the window and adding a menu

There are a few flaws with the application from Step 5. The biggest problem is that the drawing window doesn't know how to paint itself. To see this for yourself, try drawing a line in the window, minimizing the application, then restoring it. The line you drew is gone.You can find the source for Step 6 in the files STEP06.CPP and STEP06.RC in the directory EXAMPLES\OWL\TUTORIAL.

Another problem is that the only way the user can access the application is with the mouse. The user can either press the left button to draw a line or the right button to change the pen size.

In Step 6, you'll make it possible for the application to remember the contexts of the window and redraw it. You'll also add some menus to increase the number of ways the user can access the application.

Repainting the window

There are two problems that must be dealt with when you're trying to paint the window:

- There must be a way to remember what was displayed in the window.
- There must be a way to redraw the window.

Storing the drawing

In the earlier steps of the tutorial application, the line in the window was drawn as the user moved the mouse while holding the left mouse button. This approach is fine for drawing the line, but doesn't store the points in the line for later use.

Because the line is composed of a number of points in the window, you can store each point in the ObjectWindows *TPoint* class. And because each line is composed of multiple points, you need an array of *TPoint* objects to store a line. Instead of attempting to allocate, manage, and update an array of *TPoint* objects from scratch, the tutorial application uses the Borland container class *TArray* to define a data type called *TPoints*. It also uses the Borland container class *TArrayIterator* to define an iterator called *TPointsIterator*. The definitions of these two types look like this:

```
typedef TArray<TPoint> TPoints;
typedef TArrayIterator<TPoint> TPointsIterator;
```

The *TDrawWindow* class adds a *TPoints* object in which it can store the points in the line. It actually uses a *TPoints* *, a **protected** member called *Line*, which is set to point to a *TPoints* array created in the constructor. The constructor now looks something like this:

```
TDrawWindow::TDrawWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;
  PenSize = 1;
  Pen = new TPen(TColor::Black, PenSize);
  Line = new TPoints(10, 0, 10);
}
```

TPoints

The Borland C++ container class library and the *TArray* and *TArrayIterator* classes are explained in detail in Chapter 1 of the *Class Libraries Guide*. For now, here's a simple explanation of how the *TPoints* and *TPointsIterator* container classes are used in the tutorial application. To use the *TArray* and *TArrayIterator* classes, you must include the header file classlib\arrays.h.

The *TArray* constructor takes three parameters, all **int**s:

- The first parameter represents the upper boundary of the array; that is, how high the array count can go.
- The second parameter represents the lower boundary of the array; that is, the number at which the array count begins. This parameter defaults to 0, matching the C and C++ convention of starting arrays at member 0.
- The third parameter represents the array delta. The array delta is the number of members that are added when the array grows too large to contain all the members of the array.

Here's the statement that allocates the initial array of points in the *TDrawWindow* constructor:

```
Line = new TPoints(10, 0, 10);
```

The array of points is created with room for ten members, beginning at 0. Once ten objects are stored in the array, attempting to add another object adds room for ten new members to the array. This lets you start with a small conservative array size, but also

alleviates one of the main problems normally associated with static arrays, which is running out of room and having to reallocate and expand the array.

Once you've created an array, you need to be able to manipulate it. The *TArray* class (and, by extension, the *TPoints* class) provides a number of functions to add members, delete members, clear the array, and the like. The tutorial application uses only a small number of the functions provided. Here's a short description of each function:

- The *Add* function adds a member to the array. It takes a single parameter, a reference to an object of the array type. For example, adding a *TPoint* object to a *TPoints* array would look something like this:

```
// Construct a TPoints array (an array of TPoint objects)
TPoints Points(10, 0, 10);

// Construct a TPoint object
TPoint p(3,4);

// Add the TPoint object p to the array
Points.Add(p);
```

- The *Flush* function clears all the members of an array and resets the number of array members back to the initial array size. It takes no parameters. To clear the array in the previous sample code, the function call would look something like this:

```
// Clear all members in the Points array
Points.Flush();
```

- The *GetItemsInContainer* function returns the total number of items in the container. Note that this number indicates the number of actual objects added to the container, not the space available. For example, even though the container may have enough room for 30 objects, it might only contain 23 objects. In this case, *GetItemsInContainer* would return 23.

TPointsIterator

Iterators—in this case the *TPointsIterator* type—let you move through the array, accessing a single member of the array at a time. An iterator constructor takes a single parameter, a reference to a *TArray* of objects (the type of objects in the array is set up by the definition of the iterator). Here's what an iterator looks like when it's set up using the *Line* member of the *TDrawWindow* class:

```
TPointsIterator i(*Line);
```

Note that *Line* is dereferenced because the iterator constructor takes a *TPoints* & for its parameter, and *Line* is a *TPoints* *. Dereferencing the pointer makes *Line* comply with the iterator constructor type requirements.

Once you've created an iterator, you can use it to access each object in the array, one at a time, starting with the first member. In the tutorial application, the iterator isn't used very much and you won't learn much about the possibilities of an iterator from it. But the tutorial does use two properties of iterators that require a note of explanation:

- You can move through the objects in the array using the ++ operator on the iterator. This returns a reference to the current object and increments the iterator to the next object in the array. The order in which it performs these two actions depends on whether you use the ++ operator as a prefix or postfix operator. Using it as a prefix operator (for example, *++i*) increments the iterator to the next object, then returns a reference to that object. Using it as a postfix operator (for example, *i++*) returns a reference to the current object, then increments the iterator to the next object.

  When you attempt to increment the iterator past the last member of the array, the iterator is set to 0. You can use this as a test in any Boolean conditional. For example:

```
TPointsIterator i(*Line);
while(i)
  i++;
```

- You can also access the current object with the *Current* function. Calling the current function returns a reference to the current object. You can then perform operations on the object as if it were a regular instance of the object. For example, you can test a point accessed by an iterator against the value of another point:

```
TPointsIterator i(*Line);
TPoint tmp(5, 6);
if (i.Current() == tmp)
  return true;
else
  return false;
```

Using the array classes

Once the *Line* array is created in the *TDrawWindow* constructor, it is accessed in four main places:

- The *EvLButtonDown* function. The array is flushed at the beginning of the function before the screen is invalidated. The beginning point of the line is then inserted towards the end of the function. The *EvLButtonDown* function should look something like this:

```
void
TDrawWindow::EvLButtonDown(uint, TPoint& point)
{
  Line->Flush();
  Invalidate();
  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->SelectObject(*Pen);
    DragDC->MoveTo(point);
    Line->Add(point);
  }
}
```

- The *EvMouseMove* function. Each point in the line is added to the array as the user draws in the window. The *EvMouseMove* function should look something like this:

```
void
TDrawWindow::EvMouseMove(uint, TPoint& point)
{
  if (DragDC) {
    DragDC->LineTo(point);
    Line->Add(point);
  }
}
```

- The *Paint* function. This function is described in the next section.
- The *CmFileNew* function. This function is described on page 34.

Paint function

In standard C Windows programs, if you need to repaint a window manually, you catch the WM_PAINT messages and do whatever you need to do to repaint the screen. This might lead you to think that the proper way to repaint the window in the *TDrawWindow* class is to add the EV_WM_PAINT macro to the class' response table and set up a function called *EvPaint*.

You can do this if you want. However, a better way is to override the *TWindow* function *Paint*. *TDrawWindow*'s base class *TWindow* actually does quite a bit of work in its *EvPaint* function. It sets up the *BeginPaint* and *EndPaint* calls, creates a device context for the window, and so on.

Paint is a **virtual** member of the *TWindow* class. *TWindow*'s *EvPaint* calls it in the middle of its processing. The default *Paint* function doesn't do anything. You can use it to provide the special processing required to draw a line from a *TPoints* array.

Here is the signature of the *Paint* function. This is added to the *TDrawWindow* class:

```
void Paint(TDC&, bool, TRect&);
```

where:

- The first parameter is the device context set up by the calling function. This is the device context you should use when working.
- If the second parameter is true, you are supposed to clear the device context before painting the window. If it's false, you are supposed to paint over what is already contained in the window.
- The third parameter indicates the invalid area of the device context that needs to be repainted.

In the current case, you always want to clear the window. You can also assume that the entire area of the drawing needs to be repainted. The *Paint* function implements this basic algorithm:

- Create an iterator to go through the points in the line.
- Select the pen into the device context passed into the *Paint* function.
- If this is the first point in the array, set the current point to the coordinates contained in the current array member.

- While there are still points left in the array, draw lines from the current point to the point contained in the current array member.

The *TDrawWindow::Paint* function now looks something like this:

```
void
TDrawWindow::Paint(TDC& dc, bool, TRect&)
{
  bool first = true;
  TPointsIterator i(*Line);

  dc.SelectObject(*Pen);

  while (i) {
    TPoint p = i++;

    if (!first)
      dc.LineTo(p);
    else {
      dc.MoveTo(p);
      first = false;
    }
  }
}
```

Menu commands

There are a number of steps you need to perform to add a menu choice and its corresponding event handler to your application:

- Define the event identifier for the menu choice. By convention, this identifier is all capital letters, and begins with CM_. For example, the identifier for the File Open menu choice is CM_FILEOPEN.
- Add the appropriate menu resource to your resource file.
- Add an event-handling function for the menu choice to your class. The ObjectWindows 2.5 convention is to name this function the same name as the event identifier, except omitting the underscore and using initial capital letters and lowercase letters for the rest. For example, the function that handles the CM_FILEOPEN event is named *CmFileOpen*.
- Add an EV_COMMAND macro to your class' response table, associating the event identifier with the event-handling function. This macro takes two parameters; the first is the event identifier and the second is the name of the event-handling function. For example, the response table entry for the File Open menu choice looks like this:

```
EV_COMMAND(CM_FILEOPEN, CmFileOpen),
```

- The EV_COMMAND macro requires the signature of the event-handling function to take no parameters and return **void**. So the signature of the event-handling function for the File Open menu choice looks like this:

```
void CmFileOpen();
```

Adding event identifiers

You need to add identifiers for each of these menu choices. Here's the definition of the event identifiers:

```
#define CM_FILENEW    201
#define CM_FILEOPEN   202
#define CM_FILESAVE   203
#define CM_FILESAVEAS 204
#define CM_ABOUT      205
```

These identifiers are contained in the file STEP06.RC. The ObjectWindows style places the definitions of identifiers in the resource script file, instead of a header file. This cuts down on the number of source files required for a project, and also makes it easier to maintain the consistency of identifier values between the resources and the application source code.

The actual resource definitions in the resource file are contained in a block contained in an **#ifndef/#endif** block, like so:

```
#ifdef RC_INVOKED
  // Resource definitions here.
  :
#endif
```

RC_INVOKED is defined by all resource compilers, but not by C++ compilers. The resource information is never seen during C++ compilation. Identifier definitions should be placed outside this **#ifndef/#endif** block, usually at the beginning of the file.

Adding menu resources

For now, you want to add five menu choices to the application:

- File New
- File Open
- File Save
- File Save As
- Help About

Each of these menu choices needs to associated with the correct event identifier; that is, the File Open menu choice should send the CM_FILEOPEN event.

The menu resource is attached to the application in the *InitMainWindow* function. You need to call the main window's *AssignMenu* function. To get the main window, you can call the *GetMainWindow* function. The *InitMainWindow* function should look like this:

```
void InitMainWindow()
{
  SetMainWindow(new TFrameWindow(0, "Drawing Pad", new TDrawWindow));
  GetMainWindow()->AssignMenu("COMMANDS");
}
```

Adding response table entries

Each event identifier needs to be associated with its corresponding handler. To do this, add the following lines to the response table:

```
EV_COMMAND(CM_FILENEW, CmFileNew),
EV_COMMAND(CM_FILEOPEN, CmFileOpen),
EV_COMMAND(CM_FILESAVE, CmFileSave),
EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs),
EV_COMMAND(CM_ABOUT, CmAbout),
```

Adding event handlers

Now you need to add a function to handle each of the events you've just added to the response table. Because these functions will eventually grow rather large, you should declare them in the class declaration and define them outside the class declaration.

The declarations of these function should look something like this:

```
void CmFileNew();
void CmFileOpen();
void CmFileSave();
void CmFileSaveAs();
void CmAbout();
```

Implementing the event handlers

The last step in implementing the event handlers is defining the functions. For now, leave the implementation of these functions to a bare minimum. Most of them can just pop up a message box saying that the function has not yet been implemented. The functions that are set up this way are *CmFileOpen*, *CmFileSave*, *CmFileSaveAs*, and *CmAbout*. Here's how these functions look:

```
void
TDrawWindow::CmFileOpen()
{
  MessageBox("Feature not implemented", "File Open", MB_OK);
}
```

The only function that's implemented in this step is the *CmFileNew* function. That's because it's very easy to set up. All that needs to be done is to clear the array of points and erase the window. The *CmFileNew* function looks like this:

```
void
TDrawWindow::CmFileNew()
{
  Line->Flush();
  Invalidate();
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Event handling is discussed in Chapter 4 of the *ObjectWindows Programmer's Guide.*
- Window classes are discussed in Chapter 7 of the *ObjectWindows Programmer's Guide.*
- Menus and menu objects are explained in Chapter 8 of the *ObjectWindows Programmer's Guide.*
- The Borland C++ container class library and the *TArray* and *TArrayIterator* classes are explained in Chapter 1 of the *Class Libraries Guide.*

Chapter

7

Using common dialog boxes

In this step, you'll implement the event-handling functions you added in Step 6. The *CmFileOpen* function, the *CmFileSave* function, and the *CmFileSaveAs* function use the ObjectWindows classes *TFileOpenDialog* and *TFileSaveDialog*. These classes encapsulate the Windows Open and Save common dialog boxes to prompt the user for file names. You can find the source for Step 7 in the files STEP07.CPP and STEP07.RC in the directory EXAMPLES\OWL\TUTORIAL.

You'll make the *CanClose* function check whether the drawing in the window has changed before the drawing is discarded. If the drawing has changed, the user is given a chance to either save the file, continue without saving the file, or abort the close operation entirely.

Also, to implement the *CmFileOpen* function, the *CmFileSave* function, and the *CmFileSaveAs* function, you need to add two more **protected** functions, *OpenFile* and *SaveFile*, to the window class. These functions are discussed a little later in this step.

Changes to TDrawWindow

To implement the menu commands, add some new data members to the *TDrawWindow* class: *FileData*, *IsDirty*, and *IsNewFile*.

FileData

The *FileData* member is a pointer to a *TOpenSaveDialog::TData* object. The *TOpenSaveDialog* class is the direct base class of both the *TFileOpenDialog* class and the *TFileSaveDialog* class. Both of these classes use the *TOpenSaveDialog::TData* class to contain information about the current file or file operation, such as the file name, the initial directory to search, file name filters, and so on.

FileData is initialized in the *TDrawWindow* constructor to a **new**ed *TOpenSaveDialog::TData* object. Because *FileData* is a pointer to an object, a **delete**

statement must be added to the *TDrawWindow* destructor to ensure that the object is removed from memory when the application terminates.

IsDirty

The *IsDirty* flag indicates whether the current drawing is "dirty," that is, whether the drawing has been saved since it was last modified by the user. If the drawing hasn't been modified, or if the user hasn't drawn anything on an empty window, *IsDirty* is set to false. Otherwise, it is set to true. *IsDirty* is set to false in the *TDrawWindow* constructor because the drawing hasn't been modified yet.

Outside of the constructor, the *IsDirty* flag is set in a number of functions:

- In the *EvLButtonDown* function, *IsDirty* is set to true to reflect the change made to the drawing.
- In the *CmFileNew* function, *IsDirty* is set to false when the window is cleared.
- In the *OpenFile* and *SaveFile* functions, *IsDirty* is set to false to reflect that the drawing hasn't been modified since last saved or loaded.

IsNewFile

The *IsNewFile* flag indicates whether the file has a name. A file has a name if it was loaded from an existing file or has been saved to disk to some file name. If the file has a name (that is, if it's been saved previously or was loaded from an existing file), the *IsNewFile* flag is set to false. *IsNewFile* is set to true in the *TDrawWindow* constructor because the drawing hasn't yet been saved with a name.

Outside the constructor, the *IsNewFile* flag is set in a number of functions:

- In the *CmFileNew* function, *IsNewFile* is set to true when the window is cleared.
- In the *OpenFile* and *SaveFile* functions, *IsNewFile* is set to false to reflect that the drawing has been saved to disk.

Improving CanClose

The *CanClose* function that you've been using since Step 2 of this tutorial has a couple of flaws. First, whenever it's called, it prompts the user to save the drawing. This isn't necessary if the drawing hasn't been changed since it was loaded, saved, or the window was cleared. Second, a simple yes or no answer to this question isn't sufficient. For example, if the user didn't intend to close the window, the desired response is to cancel the whole operation.

Checking the *IsDirty* flag tells the *CanClose* function whether it's even necessary to prompt the user for approval of the closing operation. If the drawing isn't dirty, there's no need to ask whether it's OK to close. The user can simply reload the file.

If the file is dirty, then the *CanClose* function pops up a message box. Using the MB_YESNOCANCEL flag in the message box call gives the user three possible choices instead of two:

- Choosing Cancel means the user wants to abort the entire close operation. In this case, when *MessageBox* returns IDCANCEL, the *CanClose* function returns false, signaling to the calling function that it's *not* all right to proceed.
- Choosing Yes means that the user wants to save the file before proceeding. When *MessageBox* returns IDYES, the *CanClose* function calls the *CmFileSave* function (*CmFileSave* is explained later in this section). After calling *CmFileSave*, *CanClose* returns true, signaling to the calling function that it's all right to proceed.
- Choosing No means that the user doesn't want to save the file before proceeding. In this case, *CanClose* takes no further action and returns true.

The code for the new *CanClose* function looks something like this:

```
bool
TDrawWindow::CanClose()
{
  if (IsDirty)
    switch(MessageBox("Do you want to save?", "Drawing has changed",
                      MB_YESNOCANCEL | MB_ICONQUESTION)) {
      case IDCANCEL:
        // Choosing Cancel means to abort the close -- return false.
        return false;
      case IDYES:
        // Choosing Yes means to save the drawing.
        CmFileSave();
    }
  return true;
}
```

Note that the *CmFileNew* function is modified in this step to take advantage of the new *CanClose* function.

CmFileSave function

The *CmFileSave* function is relatively simple. It checks whether the drawing is new by testing *IsNewFile*. If *IsNewFile* is true, *CmFileSave* calls *CmFileSaveAs*, which prompts the user for a file in which to save the drawing. Otherwise, it calls *SaveFile*, which does the actual work of saving the drawing.

The *CmFileSave* function should look something like this:

```
void
TDrawWindow::CmFileSave()
{
  if (IsNewFile)
    CmFileSaveAs();
```

```
    else
      SaveFile();
}
```

CmFileOpen function

The *CmFileOpen* function is also fairly simple. It first checks *CanClose* to make sure it's OK to close the current drawing and open a new file. If the *CanClose* function returns false, *CmFileOpen* aborts.

After ensuring that it's OK to proceed, *CmFileOpen* creates a *TFileOpenDialog* object. The *TFileOpenDialog* constructor can take up to five parameters, but for this application you need to use only two. The last three parameters all have default values. The two parameters you need to provide are a pointer to the parent window and a reference to a *TOpenSaveDialog::TData* object. In this case, the pointer to the parent window is the **this** pointer. The *TOpenSaveDialog::TData* object is provided by *FileData*.

Once the dialog box object is constructed, it is executed by calling the *TFileOpenDialog::Execute* function. There are only two possible return values for the *TFileOpenDialog*, IDOK and IDCANCEL. The value that is returned depends on whether the user presses the OK or Cancel button in the File Open dialog box.

If the return value is IDOK, *CmFileOpen* then calls the *OpenFile* function, which does the actual work of opening the file. The *Execute* function also stores the name of the file the user selected into the *FileName* member of *FileData*. If the return value is not IDOK (that is, if the return value is IDCANCEL), no further action is taken and the function returns.

The *CmFileOpen* function should look something like this:

```
void
TDrawWindow::CmFileOpen()
{
  if (CanClose())
    if (TFileOpenDialog(this, *FileData).Execute() == IDOK)
      OpenFile();
}
```

CmFileSaveAs function

The *CmFileSaveAs* function can be used in two ways: to save a new drawing under a new name and to save an existing drawing under a name different from its present name.

To determine which of these the user is doing, *CmFileSaveAs* first checks the *IsNewFile* flag. If the file is new, *CmFileSaveAs* copies a null string into the *FileName* member of *FileData*. If the file is not new, *FileName* is left as it is.

The distinction between these two is quite important. If *FileName* contains a null string, the default name in the File Name box of the File Open dialog box is set to the name filter found in the *FileData* object, in this case, *.pts. But if *FileName* already contains a name, that name plus its directory path is inserted in the File Name box.

Once this has been done, *TFileSaveDialog* is created and executed. This works exactly the same as *TFileOpenDialog* does in the *CmFileOpen* function. If the *Execute* function returns IDOK, *CmFileSaveAs* then calls the *SaveFile* function.

The *CmFileSaveAs* function should look something like this:

```
void
TDrawWindow::CmFileSaveAs()
{
  if (IsNewFile)
    strcpy(FileData->FileName, "");

  if ((new TFileSaveDialog(this, *FileData))->Execute() == IDOK)
    SaveFile();
}
```

Opening and saving drawings

The *CmFileOpen*, *CmFileSave*, and *CmFileSaveAs* functions only provide the interface to let the user open and save drawings. The actual work of opening and saving files is done by the *OpenFile* and *SaveFile* functions. This section describes how these functions perform these actions, but it doesn't provide technical explanations of the entire functions.

OpenFile function

The *OpenFile* function opens the file named in the *FileName* member of the *FileData* object as an *ifstream*, one of the standard C++ iostreams. If the file can't be opened for some reason, *OpenFile* pops up a message box informing the user that it couldn't open the file and then returns.

Once the file is successfully opened, the *Line* array is flushed. *OpenFile* then reads in the number of points saved in the file, which is the first data item stored in the file. It then sets up a **for** loop that reads each point into a temporary *TPoint* object. That object is then added to the *Line* array.

Once all the points have been read in, *OpenFile* calls *Invalidate*. This invalidates the window region, causing a WM_PAINT message to be sent and the new drawing to be painted in the window.

Lastly, *OpenFile* sets *IsDirty* and *IsNewFile* both to false. The *OpenFile* function should look something like this:

```
void
TDrawWindow::OpenFile()
{
  ifstream is(FileData->FileName);

  if (!is)
    MessageBox("Unable to open file", "File Error", MB_OK | MB_ICONEXCLAMATION);
  else {
```

```
    Line->Flush();
    unsigned numPoints;
    is >> numPoints;
    while (numPoints--) {
      TPoint  point;
      is >> point;
      Line->Add(point);
    }
  }

  IsNewFile = IsDirty = false;
  Invalidate();
}
```

SaveFile function

The *SaveFile* function opens the file named in the *FileName* member of *FileData* as an *ofstream*, one of the standard C++ iostreams. If the file can't be opened for some reason, *SaveFile* pops up a message box informing the user that it couldn't open the file and then returns.

Once the file has been opened, the function *Line->GetItemsInContainer* is called. The result is inserted into the file. This number is read in by the *OpenFile* function to determine how many points are stored in the file.

After that, *SaveFile* sets up an iterator called *i* from *Line*. This iterator goes through all the points contained in the *Line* array. Each point is then inserted into the stream until there are no points left.

Lastly, *IsNewFile* and *IsDirty* are set to false. Here is how the *SaveFile* function should look:

```
void
TDrawWindow::SaveFile()
{
  ofstream os(FileData->FileName);

  if (!os)
    MessageBox("Unable to open file", "File Error",
               MB_OK | MB_ICONEXCLAMATION);
  else {
    os << Line->GetItemsInContainer();
    TPointsIterator i(*Line);
    while (i)
      os << i++;
    IsNewFile = IsDirty = false;
  }
}
```

CmAbout function

The *CmAbout* function demonstrates how easy it is to use custom dialog boxes in ObjectWindows. This function contains only one line of code. It uses the *TDialog* class and the IDD_ABOUT dialog box resource to pop up an information dialog box.

TDialog can take up to three parameters:

- The first parameter is a pointer to the dialog box's parent window. Just as with the *TFileOpenDialog* and *TFileSaveDialog* constructors, you can use the **this** pointer, setting the parent window to the *TDrawWindow* object.
- The second parameter is a reference to a *TResId* object. This should be the resource identifier of the dialog box resource.

Note Usually you don't actually pass in a *TResId* reference. Instead you pass a resource identifier number or string, just as you would for a dialog box created using regular Windows API calls. Conversion operators in the *TResId* class resolve the parameter into the proper type.

- The third parameter, a *TModule* *, usually uses its default value.

Once the dialog box object is constructed, all that needs to be done is to call the *Execute* function. Once the user closes the dialog box and execution is complete, *CmAbout* returns. The temporary *TDialog* object goes out of scope and disappears.

The code for *CmAbout* should look like this:

```
void
TDrawWindow::CmAbout()
{
  TDialog(this, IDD_ABOUT).Execute();
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- The *CanClose* function is discussed in Chapter 2 in the *ObjectWindows Programmer's Guide*.
- Dialog boxes, including the *TFileOpenDialog* and the *TFileOpenDialog* classes, are discussed in Chapter 9 in the *ObjectWindows Programmer's Guide*.

Chapter

Adding multiple lines

You can find the source for Step 8 in the files STEP08.CPP and STEP08.RC in the directory EXAMPLES\OWL\TUTORIAL. Step 8 makes a great leap in terms of usefulness. In this step, you'll add a new class, *TLine*, that is derived from the *TPoints* array you've been using to contain the points in a line. You'll then define another array class, *TLines*, that contains an array of *TLine* objects, enabling us to have multiple lines in the window. You'll add streaming operators to make it a little easier to save drawings. Lastly, you'll develop the *Paint* function further to handle drawings with multiple lines.

TLine class

The *TLine* class is derived from the public base class *TPoints*. This gives *TLine* all the functionality that you've been using with the *Line* member of the *TDrawWindow* class. This includes the *Add*, *Flush*, and *GetItemsInContainer* functions that you've been using. In addition, you can continue to use *TPointsIterator* with the *TLine* class in the same way you used it with *TPoints*.

But because you're creating your own class now, you can also add any additional functionality you need. For example, you should add a data member to contain the size of the pen for each line. Then, to hide the data, add accessor functions to manipulate the data.

In *TLine*, the pen size is contained in a **protected int** called *PenSize*. *PenSize* is accessed by one of two functions, both called *QueryPen*. Both versions of *QueryPen* return an **int**, which contains the value of *PenSize*. Here's the difference between the two functions:

- The first *QueryPen* function takes no parameters. This function returns the pen size.
- The second *QueryPen* function takes a single parameter, an **int**. This function sets *PenSize* to the value passed in, then returns the new value of *PenSize*. You can use the return value to check whether *QueryPen* actually set the pen to the value you passed to it. This version of *QueryPen* checks the value of the parameter to make sure that it's a legal value for the pen size.

TLine also contains a definition for the == operator. This operator checks to see if the two objects are actually the same object. If so, the operator returns true. Defining an array using the *TArray* class (which you'll do later when defining *TLines*) requires that the object used in *TArray* have the == operator defined.

Lastly you should declare two operators, << and >>, to be **friend**s of the *TLine* class. When these operators are implemented later in this section, they'll provide easy access to stream operations for the *SaveFile* and *OpenFile* functions.

Here is the declaration of the *TLine* class:

```
class TLine : public TPoints
{
  public:
    TLine(int penSize = 1) : TPoints(10, 0, 10) { PenSize = penSize; }

    int QueryPen() const { return PenSize; }
    int QueryPen(int penSize);

    // The == operator must be defined for the container class,
    // even if unused
    bool operator ==(const TLine& other) const
      { return &other == this; }
    friend ostream& operator <<(ostream& os, const TLine& line);
    friend istream& operator >>(istream& is, TLine& line);

  protected:
    int PenSize;
};
```

TLines array

Once you've defined the *TLine* class, you can define the *TLines* array and the *TLinesIterator* array. These containers work the same way as the *TPoints* and *TPointsIterator* container classes that you defined earlier. The only difference is that, instead of containing an array of *TPoint* objects like *TPoints*, *TLines* contains an array of *TLine* objects.

Here are the definitions of *TLines* and *TLinesIterator*:

```
typedef TArray<TLine> TLines;
typedef TArrayIterator<TLine> TLinesIterator;
```

Insertion and extraction of TLine objects

Most objects that need to be saved to and retrieved from files on a regular basis are set up to use the insertion and extraction operators << and >>. By declaring these operators as friends of *TLine*, you need to define the operators to handle the particular type of data encapsulated in *TLine*.

Having these operators defined gives you the ability to place an entire *TLine* object into a file with a single line of code. You'll see how this is used when you make the changes to the *OpenFile* and *SaveFile* functions.

Insertion operator <<

In essence, the insertion operator takes on the functionality of the *SaveFile* function used in Step 7. It doesn't have to open a file (that's handled by whatever function uses the operator) and it has an extra piece of data to insert (*PenSize*). Other than that, it's not much different. Compare the definition of this function with the *SaveFile* function from Step 7. Notice the use of *TPointsIterator* with the *TLine* object:

```
ostream& operator <<(ostream& os, const TLine& line)
{
  // Write the number of points in the line
  os << line.GetItemsInContainer() << '

  // Write the pen size
  os << ' ' << line.PenSize;

  // Get an iterator for the array of points
  TPointsIterator j(line);

  // While the iterator is valid (i.e. it hasn't run out of points)
  while(j)
    // Write the point from the iterator and increment the array.
    os << j++;

  os << '

  // return the stream object
  return os;
}
```

Extraction operator >>

Much like the insertion operator, the extraction operator takes on the functionality of the *OpenFile* function in Step 7. It doesn't have to open a file itself and it has an extra piece of data to extract. Other than that, it's implemented similarly to the *OpenFile* function:

```
istream& operator >>(istream& is, TLine& line)
{
  unsigned numPoints;

  is >> numPoints;

  is >> line.PenSize;

  while (numPoints--) {
    TPoint point;
    is >> point;
```

```
    line.Add(point);
  }
  // return the stream object
  return is;
}
```

Extending TDrawWindow

There are a number of changes required in *TDrawWindow* to accommodate the new *TLine* class. First there are a number of changes in data members:

- *PenSize* is removed. Each individual line now contains its pen size.
- The *Line* data member is changed from a *TPoints* * to a *TLine* *. The *Line* object holds the points in the line currently being drawn.
- The *Lines* data member, a *TLines* *, is added. The *Lines* object contains all the *TLine* objects.

There are also a number of functions that are modified or added:

- The *SetPenSize* function is made **protected** because changes to the pen size should be made to the *TLine* class. *SetPenSize* should now be used only by the *TDrawWindow* class internally. *SetPenSize* also sets the pen size for the current line by calling that line's *QueryPen* function.
- The *GetPenSize* function is added. This function implements the *TInputDialog* that was handled in *EvRButtonDown*. This is because two functions now use this same dialog box, *EvRButtonDown* and *CmPenSize*.
- The *EvRButtonDown* function now calls *GetPenSize* to open the input dialog box.
- The *CmPenSize* function handles the CM_PENSIZE event. This event comes from a new menu choice, Pen Size, on a new menu, Tools. This function is added to give the user another way to change the pen size.
- The *OpenFile* and *SaveFile* functions are modified to store an array of *TLine* objects instead of an array of *TPoint* objects. By using the insertion and extraction operators, these functions change very little from their prior forms.

In addition, the *Paint* function is changed quite a bit, as described in the following section.

Paint function

The *Paint* function must now perform two iterations instead one. Instead of iterating through a single array of points, *Paint* must now iterate through an array of lines. For each line, it must set the pen width and then iterate through the points that compose the line.

Paint does this by first creating an iterator from *Lines*. This iterator goes through the array of lines. For each line, *Paint* queries the pen size of the current line. It sets the

window's *Pen* to this size and selects this pen into the device context. It then creates an iterator for the current line and increments the line array iterator.

The next part of *Paint* looks like the *Paint* function from Step 7. That's because it does basically the same thing as that function—it takes the array of points and draws the line in the window.

Here is the code for the new *Paint* function:

```
void
TDrawWindow::Paint(TDC& dc, bool, TRect&)
{
  // Iterates through the array of line objects.
  TLinesIterator i(*Lines);

  while (i) {
    // Set pen for the dc to current line's pen.
    TPen pen(TColor::Black, i.Current().QueryPen());
    dc.SelectObject(pen);

    // Iterates through the points in the line i.
    TPointsIterator j(i++);
    bool first = true;

    while (j) {
      TPoint p = j++;

      if (!first)
        dc.LineTo(p);
      else {
        dc.MoveTo(p);
        first = false;
      }
    }
  }
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Window classes are discussed in Chapter 7 of the *ObjectWindows Programmer's Guide*.
- The Borland C++ container class library and the *TArray* and *TArrayIterator* classes are explained in Chapter 1 of the *Class Libraries Guide*.

Chapter

9

Changing pens

You can find the source for Step 9 in the files STEP09.CPP and STEP09.RC in the directory EXAMPLES\OWL\TUTORIAL. In Step 9, you'll add a *TColor* member to the *TLine* class, letting the user draw with lines of different widths *and* different colors. To change the color of the line, you'll add the *CmPenColor* function. This function handles the CM_PENCOLOR menu command. *CmPenColor* uses the *TChooseColorDialog* class to let the user change colors. It also adds some helper functions to deal with changes to the width and color and give external classes access to information about the line.

Along with adding color to the pen, Step 9 adds functionality to the streaming operators to deal with the new attributes of the *TLine* class. It also adds a *Draw* function to the *TLine* class to make the class more self-sufficient and to make the *Paint* function simpler.

Changes to the TLine class

A number of changes to the *TLine* class declaration are required to accommodate the new functionality:

- There is a new **protected** data member, *Color* (a *TColor* object). *Color* and *PenSize* make up the attributes necessary to construct a *TPen* object.
- The constructor signature has changed from

  ```
  TLine(int penSize = 1);
  ```

  to

  ```
  TLine(const TColor &color = (TColor) 0, int penSize = 1);
  ```

  The constructor itself changes to set *PenSize* to the constructor's second parameter and to create a new *TPen* object and assign it to *Pen*. If no parameters are specified and the first parameter takes on its default value, *TColor::Black* is used as the pen color.

- The two *QueryPen* functions are abandoned in favor of three new functions: *QueryPenSize*, which returns the pen size as an **int**, *QueryColor*, which returns the pen color as a *TColor*, and *QueryPen*, which returns the pen as a *TPen*.
- Instead of using the query functions to set the pen attributes, there are two new functions called *SetPen*. One takes a single **int** parameter and the other takes a *TColor* **&** and two **int**s. The pen query and set functions are discussed in the next section.
- A *Draw* function is added so that the *TLine* class dictates how it is drawn. This function is **virtual** so that it can be easily overridden in a derived class.

Here's how the new *TLine* class declaration should look:

```
class TLine : public TPoints {
  public:
    // Constructor to allow construction from a color and a pen size.
    // Also serves as default constructor.
    TLine(const TColor &color = TColor(0), int penSize = 1)
      : TPoints(10, 0, 10), PenSize(penSize), Color(color) {}

    // Functions to modify and query pen attributes.
    int QueryPenSize() { return PenSize; }
    TColor& QueryColor() { return Color; }
    void SetPen(TColor &newColor, int penSize = 0);
    void SetPen(int penSize);

    // TLine draws itself. Returns true if everything went OK.
    virtual bool Draw(TDC &) const;

    // The == operator must be defined for the container class,
    // even if unused
    bool operator ==(const TLine& other) const
      { return &other == this; }
    friend ostream& operator <<(ostream& os, const TLine& line);
    friend istream& operator >>(istream& is, TLine& line);

  protected:
    int PenSize;
    TColor Color;
};
```

Pen access functions

In Step 8, the *QueryPen* function could be used both to access the current size of the pen and to set the size of the pen. The new *TLine* query functions—*QueryPenSize* and *QueryColor*—can't be used to modify the pen attributes. These functions only return pen attributes.

To set pen attributes, there are two new functions called *SetPen*. The first *SetPen* sets just the pen size. The other *SetPen* can be used to set the color, size, and style of the pen. But by letting the second and third parameters take on their default values, you can use the second constructor to set just the color. Here's the code for these functions:

```
void
TLine::SetPen(int penSize)
{
  if (penSize < 1)
    PenSize = 1;
  else
    PenSize = penSize;
}

void
TLine::SetPen(TColor &newColor, int penSize)
{
  // If penSize isn't the default (0), set PenSize to the new size.
  if (penSize)
    PenSize = penSize;

  Color = newColor;
}
```

Draw function

The *Draw* function draws the line in the window, taking that functionality from the window's *Paint* function. This functionality is moved because the *TLine* object can now dictate how it gets painted onscreen. Take a look at the code for the *Draw* function below and compare this to the *Paint* function from Step 8. From a certain point, the two bits of code are nearly identical:

```
bool
TLine::Draw(TDC &dc) const
{
  // Set pen for the dc to the values for this line
  TPen pen(Color, PenSize);
  dc.SelectObject(pen);

  // Iterates through the points in the line i.
  TPointsIterator j(*this);
  bool first = true;

  while (j) {
    TPoint p = j++;

    if (!first)
      dc.LineTo(p);
    else {
      dc.MoveTo(p);
      first = false;
    }
  }
  dc.RestorePen();
  return true;
}
```

After putting all this code into the *TLine* class, the *TDrawWindow::Paint* function is greatly simplified:

```
void
TDrawWindow::Paint(TDC& dc, bool, TRect&)
{
  // Iterates through the array of line objects.
  TLinesIterator i(*Lines);

  while (i)
    i++.Draw(dc);
}
```

Insertion and extraction operators

There also some changes to the insertion and extraction operators that are necessary to handle the revised *TLine* class.

- The insertion operator is modified to write out the *PenSize* and *Color* member. It then writes out the points just as it did before.
- The extraction operator reads in the data and uses the *PenSize* and *Color* data in the *SetPen* function. Each point is read in from the file and added to the object.

Changes to the TDrawWindow class

There are a few fairly minor changes to the *TDrawWindow* class to accommodate the revised *TLine* class:

- The *Pen* data member is constructed from the size and color of the current line.
- The *SetPenSize* function is removed. The function *GetPenSize* opens a *TInputDialog* for the user to enter a new pen size in. *GetPenSize* then calls the function *Line->SetPen* to actually set the pen size.
- The *CmPenColor* function is added to handle the CM_PENCOLOR event. This event is sent from the new Tools menu choice Pen Color.

CmPenColor function

The *CmPenColor* function opens a *TChooseColorDialog* for the user to select a color from. Like *TFileOpenDialog* and *TFileSaveDialog*, *TChooseColorDialog* is an encapsulation of one of the Windows common dialog boxes.

Also like *TFileOpenDialog* and *TFileSaveDialog*, the *TChooseColorDialog* constructor can take up to five parameters, but in this case you need only two. The last three all have default values. The two parameters you need to provide are a pointer to the parent window and a reference to a *TChooseColorDialog::TData* object. In this case, the pointer to the parent window is simply the **this** pointer. The *TChooseColorDialog::TData* object is provided by *colors*.

Setting the *Color* member of *colors* to a particular color makes that color (or its closest equivalent displayed in the dialog box) the default color in the dialog box. By setting *Color* to the color of the current pen, you ensure that the Color dialog box reflects the current state of the application.

Setting the *CustColors* member of the *colors* object to some array of *TColor* objects sets those colors in the Custom Colors section of the Color dialog box. You can use whatever colors you want for the *CustColors* array. The values that are used in the tutorial produce a range of monochrome colors that goes from black to white.

Creating and executing a *TChooseColorDialog* works exactly the same as for a *TFileOpenDialog* or *TFileSaveDialog*. Although the Color dialog box has an extra button (the Define Custom Colors button), that button is handled by the Windows part of the common dialog box. Therefore there are only two possible results for the *Execute* function, IDOK and IDCANCEL. If the user selects Cancel, you ignore any changes from the dialog box.

On the other hand, if the user selects OK, you need to change the pen color to the new color chosen by the user. The *TChooseColorDialog* places the color chosen by the user into the *Color* member of the *colors* object. *Color* is a *TColor*, which fits nicely into the *SetPen* function of a *TLine* object.

Here's the code for the *CmPenColor* function:

```
void
TDrawWindow::CmPenColor()
{
  TChooseColorDialog::TData colors;
  static TColor custColors[16] =
  {
    0x010101L, 0x101010L, 0x202020L, 0x303030L,
    0x404040L, 0x505050L, 0x606060L, 0x707070L,
    0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
    0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
  };

  colors.Flags = CC_RGBINIT;
  colors.Color = TColor(Line->QueryColor());
  colors.CustColors = custColors;
  if (TChooseColorDialog(this, colors).Execute() == IDOK)
    Line->SetPen(colors.Color);
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- The *TPen* and *TColor* classes are discussed in Chapter 14 in the *ObjectWindows Programmer's Guide*.
- Dialog boxes, including the *TChooseColorDialog* class, are discussed in Chapter 9 in the *ObjectWindows Programmer's Guide*.

Chapter

10

Adding decorations

The only changes in Step 10 are in the *InitMainWindow* function. But these changes let you make your application more attractive and easier and more intuitive to use. In this step, you'll add a control bar with bitmap button gadgets and a status bar that displays the current menu choice. You can find the source for Step 10 in the files STEP10.CPP and STEP10.RC in the directory EXAMPLES\OWL\TUTORIAL.

There are four main changes in this step:

- Changing the main window from a *TFrameWindow* to a *TDecoratedFrame*.
- Creating a status bar and inserting it into the decorated frame window.
- Creating a control bar, along with its button gadgets, and inserting it into the decorated frame.
- Adding resources, such as a string table (which provides descriptions of each of the available menu choices) and bitmaps for the button gadgets.

Changing the main window

Changing from a *TFrameWindow* to a *TDecoratedFrame* is quite easy. Because *TDecoratedFrame* is based on *TFrameWindow*, a decorated frame can be used just about anywhere that a regular frame window is used. In this case, just create a *TDecoratedFrame* and pass it as the parameter to the *SetMainWindow* function.

Even the constructors of the *TFrameWindow* and *TDecoratedFrame* are alike. The only difference is the fourth parameter, which wasn't being used anyway. The fourth parameter for *TFrameWindow* is a **bool** that tells the frame window whether it should shrink to the size of its client window.

The fourth parameter for *TDecoratedFrame* is also a **bool**. This parameter indicates whether the decorated frame should track menu selections. Menu tracking displays a text description of the currently selected menu choice or button in a message bar or status bar. If you specify **true** for this parameter, you *must* supply a message or status

bar for the window. If you don't, your application will crash the first time it tries to send a message to the message or status bar.

If you're using a status bar, you must include the resources for it in your resource file. These resources are contained in the file STATUSBA.RC in the INCLUDE\OWL directory.

The only other difference is that the decorated frame requires some preparation, such as adding decorations like the control bar and status bar, before it can become the main window. So instead of constructing and setting the window in one step, you must construct the window, prepare it, then set it as the main window.

Creating the status bar

Status bars are created using the *TStatusBar* class. *TStatusBar* is based on the *TMessageBar* class, which is itself based on *TGadgetWindow*. Both message bars and status bars display text messages. But status bars have more options than message bars. For example, you can have multiple text gadgets, styled borders, and mode indicators (such as Insert or Overwrite mode) in a status bar.

The *TStatusBar* constructor takes five parameters, although you only use the first two. The rest of the parameters take on their default values:

- The first parameter is a pointer to the status bar's parent window. In this case, use *frame*, which is the pointer to the decorated frame window constructed earlier.
- The second parameter is a *TGadget::TBorderstyle* **enum**. It can be one of *None*, *Plain*, *Raised*, *Recessed*, or *Embossed*. This parameter determines the style of the status bar. This parameter defaults to *Recessed*.
- The third parameter is a *TModeIndicator* **enum**. It determines the keyboard modes that the status bar should show. These indicators can be one or more of *ExtendSelection*, *CapsLock*, *NumLock*, *ScrollLock*, *Overtype*, and *RecordingMacro*. This parameter defaults to 0, meaning to indicate no keyboard modes.
- The fourth parameter is a *TFont* *. This contains the font that should be used in the status bar. This defaults to *TGadgetWindowFont*.
- The fifth parameter is a *TModule* *. It defaults to 0.

Here is the status bar constructor:

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

Once the status bar is created, it is ready to be inserted into the decorated frame. This is described on page 61.

Creating the control bar

Creating the control bar is more involved than creating the status bar. You first construct the actual *TControlBar* object. Then you create the gadgets that make up the controls on the bar and insert them into the control bar.

Constructing TControlBar

The *TControlBar* constructor takes four parameters, although you need to use only the first parameter here. The rest of the parameters take on their default values:

- The first parameter is a pointer to the parent window. As with the status bar, use *frame* here to make the decorated frame the control bar's parent.
- The second parameter is a *TTileDirection* **enum**. A *TTileDirection* **enum** can have two values, *Horizontal* and *Vertical*. This tells the control bar which way to tile its controls. This parameter defaults to *Horizontal*.
- The third parameter is a *TFont* *. This contains the font that should be used in the status bar. This defaults to *TGadgetWindowFont*.
- The fourth parameter is a *TModule* *. It defaults to 0.

Here is the control bar constructor:

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

Building button gadgets

Button gadgets are used as control bar buttons. They associate a bitmap button with an event identifier. When the user presses a button gadget, it sends that event identifier. You can set this up so that pressing a button on the control is just like making a choice from a menu. In this section, you'll see how to set up buttons to replicate each of your current menu choices.

Button gadgets are created using the *TButtonGadget* class. The *TButtonGadget* constructor takes six parameters, of which you need to use only the first three:

- The first parameter is a reference to a *TResId* object (see the note on page 43 regarding the *TResId* class). This should be the resource identifier of the bitmap you want on the button. There are no real restrictions on the size of the bitmap you can use in a button gadget. There are, however, practical considerations: the control bar height is based on the size of the objects contained in the control bar. If your bitmap is excessively large, the control bar will be also.
- The second parameter is the gadget identifier for this button gadget. Usually the gadget identifier, event identifier, and bitmap resource identifier are the same. For example, the button gadget for the File New command uses a bitmap resource called CM_FILEOPEN, has the gadget identifier CM_FILEOPEN, and posts the event CM_FILEOPEN.

  The bitmap is given the same identifier in the resource file as the event identifier. This makes it a little easier on you when working with the code. This is *not* a rule, however, and you can name the bitmap and event identifier whatever you like. The only stipulation is that the event identifier must be defined and have some sort of processing enabled and the resource identifier must be valid.

  You should also notice that there are a number of entries in the application's string resource table that have the same IDs as the gadgets and events. When a string exists

with the same identifier as a button gadget, that string is displayed in the status bar when the gadget is pressed.

- The third parameter is a *TType* **enum**. This indicates what type of button this is. There are three possible button types, *Command*, *Exclusive*, and *NonExclusive*. In this application, all the buttons are command buttons. This parameter defaults to *Command*.
- The fourth parameter is a **bool** indicating whether the button is enabled. By default this parameter is **false**.
- The fifth parameter is a *TState* **enum**. This parameter indicates the initial state of the button, and can be *Up*, *Down*, or *Indeterminate*. This parameter defaults to *Up*.
- The sixth parameter is a **bool** that indicates the repeat state of the button. If the repeat state is **true**, the button repeats when it is pressed and held. By default, this parameter is **false**.

Separator gadgets

There is another type of gadget commonly used when constructing control bars, called a separator gadget. Normally gadgets in a control bar are right next to each other. A separator gadget provides a little bit of space between two gadgets. This lets you separate gadgets into groups, place them in predetermined spots on the control bar, and so on.

Separator gadgets are contained in the *TSeparatorGadget* class. This is a simple class that takes a single **int** parameter. By default the value of this parameter is 6. This parameter indicates the number of pixels of space the separator gadget should take up.

Inserting gadgets into the control bar

Once your gadgets are constructed, you need to insert them into the control bar. The control bar can take gadgets because it is derived from the class *TGadgetWindow*. *TGadgetWindow* provides the basic functionality that lets you use gadgets in a window. *TControlBar* refines that functionality, producing a control bar.

You can insert gadgets into the control bar using the *Insert* function. This version of the *Insert* function is inherited by *TControlBar* from *TGadgetWindow* (later you'll use another version of this function contained in *TDecoratedFrame*). This function takes three parameters, although you need to use only the first parameter in the tutorial application:

- The first parameter is a reference to a *TGadget* or *TGadget*-derived object.
- The second parameter is a *TPlacement* **enum**, which can have a value of *Before* or *After*. This parameter indicates whether the gadget should be placed before or after the gadget's sibling. The default value is *After*. This parameter has no effect if there is no sibling specified.
- The gadget's sibling is specified by the third parameter, which is a *TGadget* *. The sibling should have already been inserted into the control bar. This parameter defaults to 0.

In the tutorial application, constructing the gadgets and inserting them into the control bar is accomplished in a single step. Here is the code where the gadgets are inserted into the control bar:

```
cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS,
           TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
           TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT,
           TButtonGadget::Command));
```

Notice that the button gadgets replicate the menu commands you already have. This provides an easy way for the user to access frequently used menu commands. Of course, you aren't restricted to using gadgets in a control bar as substitutes or shortcuts for menu commands. Using the *TType* parameter, you can set up gadgets on a control bar to work like radio buttons (by using *Exclusive* with a group of gadgets), check boxes (using *NonExclusive*), and so on.

Inserting objects into a decorated frame

Now that you've constructed the decorations for your *TDecoratedFrame* window, all you need to do is insert the decorations into the window and make the window the main window.

Inserting decorations into a decorated frame is similar to inserting gadgets into a control bar. The *TDecoratedFrame::Insert* function takes two parameters:

- The first is a reference to a *TWindow* or *TWindow*-derived object. This *TWindow* object is the decoration. In this case, the *TWindow*-derived objects are the *TStatusBar* object and the *TControlBar* object.
- The second parameter is a *TLocation* **enum**. This parameter can have one of four values, *Top*, *Bottom*, *Left*, or *Right*. This indicates where in the decorated frame the gadget is to be placed.

Here is the code for inserting the decorations into the decorated frame:

```
// Insert the status bar and control bar into the frame
frame->Insert(*sb, TDecoratedFrame::Bottom);
frame->Insert(*cb, TDecoratedFrame::Top);
```

Once you've inserted the decorations into the frame, the last thing you have to do is set the main window to *frame* and set up the menu:

```
// Set the main window and its menu
SetMainWindow(frame);
GetMainWindow()->AssignMenu("COMMANDS");
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- Decorated frame windows are discussed in Chapter 7 in the *ObjectWindows Programmer's Guide*.
- Gadgets are discussed in Chapter 12 in the *ObjectWindows Programmer's Guide*.
- Status bars and control bars are discussed in both Chapter 7 and Chapter 12 in the *ObjectWindows Programmer's Guide*.

Chapter

11

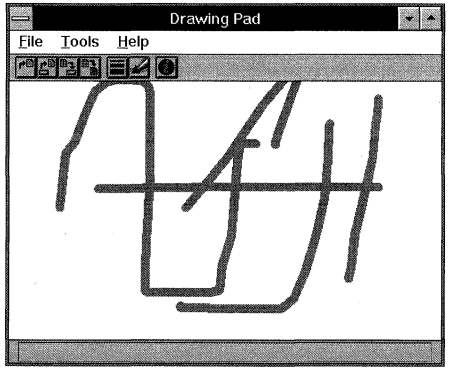
Moving to MDI

This chapter describes how to convert the application created in Step 10 to use the Multiple Document Interface, or MDI for short. The application in Step 10 is what is known as a Single Document Interface, or SDI, application. That means the application can support and display only a single document at a time.

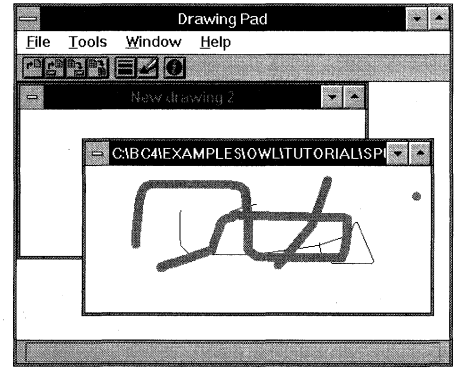
In the sense that it's used here, document doesn't have the same meaning you might be used to. Instead of a paper document or a word-processing document, a document refers to any set of data that your application displays and manipulates. In the case of the tutorial application, documents are the drawing files that the application creates. Converting the application to use MDI adds the ability to support multiple drawings open at the same time in multiple child windows. Figure 11.1 shows the difference between the SDI version of the Drawing Pad application and the MDI version that you'll produce in this step.

Figure 11.1 SDI versus MDI Drawing Pad application

SDI version



MDI version



Understanding the MDI model

An MDI application functions a little differently from an SDI application. In Step 10, the Drawing Pad application displayed a single drawing in a window. The window that actually displayed the drawing was a client of the frame window. The frame window managed general application tasks, such as menu handling, resizing, painting menus and control bars, and so on. The client window managed tasks specific to the application, such as handling mouse movements and button clicks in the client area, painting the lines in the drawing, responding to application-specific events, and so on.

In comparison, MDI applications divide tasks up three ways instead of two:

- The frame window functions much as it does in the SDI application, handling basic application functionality.
- The client window handles tasks related to creating, managing, and closing MDI child windows, along with any related functions. For example, the client window might manage the File | Open command since, in order to open an MDI child window, you usually need something to display in it.
- MDI child windows display the data in an MDI application and give the user the ability to manipulate and control the data. These windows handle application-specific tasks, much like the client window did Step 10.

In this step, you'll take the example from Step 10 and restructure it to support MDI functionality. It's not as complicated as it may seem; most of the new classes you'll construct can be taken straight from the existing *TDrawWindow* class!

Adding the MDI header files

There are a number of new header files you need to include to add MDI capability to your application. This section describes the header files that need to be changed or added. It also describes the classes that are defined in each header file.

Changing the resource script file

You need to change the include statement for the STEP10.RC resource script file to include the STEP11.RC resource script file. There are only two changes you need to make to STEP11.RC:

- Include the resource header file owl\mdi.rh.
- Add a pop-up menu called Window between the Tools menu and the Help menu. This menu should have four items, described in Table 11.1.

Table 11.1 MDI Window menu items and identifiers

Menu item text	Command identifier
Cascade	CM_CASCADECHILDREN
Tile	CM_TILECHILDREN

Table 11.1 MDI Window menu items and identifiers (continued)

Menu item text	Command identifier
Arrange Icons	CM_ARRANGEICONS
Close All	CM_CLOSECHILDREN

The functions that handle these events are described later on page 72.

Replacing the frame window header file

In the place of owl\decframe.h, you need to include owl\decmdifr.h. This header file contains the definition of the *TDecoratedMDIFrame* class, which is derived from *TMDIFrame* and *TDecoratedFrame*. *TMDIFrame*, defined in the owl\mdi.h header file, adds the support for containing an MDI client window to the support already provided by *TFrameWindow* for command processing and keyboard navigation. MDI client windows are discussed on page 65. As shown in the previous step of the tutorial, *TDecoratedFrame* provides the ability to support decorations such as control bars and status bars. Since the tutorial application already supports decorations from the previous step, you can use the decorated version of the MDI frame window to keep this functionality.

Adding the MDI client and child header files

You need to add the owl\mdi.h and owl\mdichild.h header files. owl\mdi.h contains the definition of the *TMDIFrame* and *TMDIClient* classes. *TMDIClient* provides the functionality necessary for managing MDI child windows. MDI child windows are the windows that the user of your application actually works with and that display the data contained in each document. *TMDIClient* provides the ability to

- Close all of the open MDI child windows
- Find the active MDI child window
- Initialize a new MDI child object
- Create a new MDI child window
- Arrange and manage MDI child windows, including arranging icons for minimized child windows and cascading or tiling open child windows

owl\mdichild.h contains the definition of the *TMDIChild* class, which is derived from *TWindow*. *TMDIChild* overrides a number of *TWindow*'s function to provide the ability to function as an MDI child.

You usually derive new classes from both *TMDIClient* and *TMDIChild* to provide the specific functionality required by your application. Creating new classes from *TMDIClient* and *TMDIChild* to support the Drawing Pad application is discussed later in this step.

Changing the frame window

The first step in moving the drawing application to MDI is to change the frame window. MDI applications use specialized MDI frame windows. As discussed earlier, ObjectWindows provides two MDI frame window classes, *TMDIFrame* and *TDecoratedMDIFrame*. Because we're using the *TDecoratedMDIFrame* class for the frame window, discussion of the *TMDIFrame* class is left for Chapter 7 of the *ObjectWindows Programmer's Guide*.

Here's the constructor for *TDecoratedMDIFrame*:

```
TDecoratedMDIFrame(const char far* title,
                   TResId menuResId,
                   TMDIClient& clientWnd = *new TMDIClient,
                   bool trackMenuSelection = false,
                   TModule* module = 0);
```

where:

- *title* is the caption for the frame window.
- *menuResId* is the resource identifier for the frame window's main menu.
- *clientWnd* is the MDI client window for the frame window.
- *trackMenuSelection* indicates whether this frame should track menu selections. This is the same thing as menu tracking for the *TDecoratedFrame* you constructed in the last step.
- *module* is a pointer to an program module. *module* is used to initialize the *TWindow* base object.

Besides adding the owl\decmdifr.h header file, two other changes are required to use a *TDecoratedMDIFrame* in the tutorial application. The first is changing the line in the *TDrawApp::InitMainWindow* function where the frame window is created:

```
TDecoratedMDIFrame *frame = new TDecoratedMDIFrame("Drawing Pad",
                                                   TResId("COMMANDS"),
                                                   *new TDrawMDIClient,
                                                   true);
```

As before, the frame window caption is Drawing Pad. The frame window is initialized with the COMMANDS menu resource. The client window is a new *TDrawMDIClient*, which is a *TMDIClient*-derived class that you'll define a little bit later in this step. The final parameter indicates that menu tracking should be on for this window. The *module* parameter is left to its default value of 0.

The second change is removing the *AssignMenu* call at the end of the *InitMainWindow* function of Step 10. This call is no longer necessary because the menu resource is set up by the second parameter of the *TDecoratedMDIFrame* constructor.

Your *InitMainWindow* function should now look something like this:

```
void
TDrawApp::InitMainWindow()
{
```

```
  // Create a decorated MDI frame
  TDecoratedMDIFrame *frame = new TDecoratedMDIFrame("Drawing Pad",
                                                     TResId("COMMANDS"),
                                                     *new TDrawMDIClient,
                                                     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_FILENEW, CM_FILENEW, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS, TButtonGadget::Command));
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR, TButtonGadget::Command));
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT, TButtonGadget::Command));

  // 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);
}
```

These are the only changes necessary to the *TDrawApp* class to support MDI functionality.

Creating the MDI window classes

The functionality contained in the *TDrawWindow* class in the previous step needs to be divided up into two classes in the MDI model. The reason for this is that there are two windows that handle messages and user input:

- MDI client window are created during the construction of the MDI frame class. This window is open as long as the frame window is still open (in this case, for the life of the application). This window handles the CM_FILEOPEN, CM_FILENEW, and CM_ABOUT commands.

  When the application is first started up, or when there are no drawings open, the only commands that make sense are opening drawing files, creating new drawings, and opening the About... dialog box. Other commands available in the tutorial application, such as saving drawings, changing the pen size or color, and so on, apply to a particular drawing, which must already be open and displayed in a child window.

- MDI child windows are created by the MDI client window in response to CM_FILENEW or CM_FILEOPEN commands handled by the client window. In the

tutorial application, MDI child windows handle the events handled by *TDrawWindow* in Step 10 that aren't handled by *TDrawMDIClient*:

- WM_LBUTTONDOWN
- WM_RBUTTONDOWN
- WM_MOUSEMOVE
- WM_LBUTTONUP
- CM_FILESAVE
- CM_FILESAVEAS
- CM_PENSIZE
- CM_PENCOLOR

Note that each of these commands pertains to a specific drawing or window; that is, each event only makes sense in the context of an open drawing contained in a child window. For example, in order for the user of the application to save a drawing, there must already be a drawing open. Contrast this to the events handled by the MDI client window, which either open a new child window containing a new or existing drawing or are independent of a drawing altogether.

The next sections discuss how to create the MDI client and child window classes for the tutorial application.

Creating the MDI child window class

You need to create a class declaration for the *TDrawMDIChild* class, along with defining the functions for the class. You can reuse most of the class declaration for *TDrawWindow* from Step 10, along with most of the functions with only a few changes.

Declaring the TDrawMDIChild class

The class declaration for *TDrawMDIChild* is very similar to the declaration of the *TDrawWindow* class from Step10. Here are the changes you need to make:

- Change all occurrences of *TDrawWindow* to *TDrawMDIChild*. This includes the name of the destructor, which otherwise doesn't change.
- Remove the *CmFileNew*, *CmFileOpen*, and *CmAbout* functions from the class declaration.
- The constructor for *TMDIChild* requires a *TMDIClient* reference in place of *TDrawWindow*'s *TWindow* *. This parameter indicates the parent of the MDI child window. In this case, you want to add a *TDrawMDIClient* reference to the constructor and pass this to the *TMDIChild* constructor. In addition, you should add a **const char*** for the MDI child window's caption.
- In the response table, remove the entries for handling the CM_FILENEW, CM_FILEOPEN, and CM_ABOUT events.

Your class declaration should look something like this:

```
class TDrawMDIChild : public TMDIChild {
  public:
    TDrawMDIChild(TDrawMDIClient& parent, const char* title = 0);
```

```
    ~TDrawMDIChild() { delete DragDC; delete Line; delete Lines; delete FileData; }

  protected:
    TDC *DragDC;
    TPen *Pen;
    TLines *Lines;
    TLine *Line; // To hold a single line at a time that later gets
                 // stuck in Lines
    TOpenSaveDialog::TData
              *FileData;
    bool IsDirty, IsNewFile;

    void GetPenSize(); // GetPenSize always calls Line->SetPen().

    // Override member function of TWindow
    bool CanClose();

    // Message response functions
    void EvLButtonDown(uint, TPoint&);
    void EvRButtonDown(uint, TPoint&);
    void EvMouseMove(uint, TPoint&);
    void EvLButtonUp(uint, TPoint&);
    void Paint(TDC&, bool, TRect&);
    void CmFileSave();
    void CmFileSaveAs();
    void CmPenSize();
    void CmPenColor();
    void SaveFile();
    void OpenFile();

    DECLARE_RESPONSE_TABLE(TDrawMDIChild);
};

DEFINE_RESPONSE_TABLE1(TDrawMDIChild, TWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs),
  EV_COMMAND(CM_PENSIZE, CmPenSize),
  EV_COMMAND(CM_PENCOLOR, CmPenColor),
END_RESPONSE_TABLE;
```

Creating the TDrawMDIChild functions

Just about all of the functions in *TDrawMDIChild* can be carried over from the *TDrawWindow* class. The only thing you need to do is change the class identifier in the function declarations from *TDrawWindow* to *TDrawMDIChild*. For example, the declaration for the *EvLButtonDown* function changes from this:

```
void
TDrawWindow::EvLButtonDown(uint, TPoint& point)
```

```
{
  ⋮
}
```

to this:

```
void
TDrawMDIChild::EvLButtonDown(uint, TPoint& point)
{
  ⋮
}
```

Change the class identifiers for the following functions:

GetPenSize	*CanClose*
EvLButtonDown	*EvRButtonDown*
EvMouseMove	*EvLButtonUp*
Paint	*CmFileSave*
CmFileSaveAs	*CmPenSize*
CmPenColor	*SaveFile*
OpenFile	

There is one minor change you need to make to the *CmFileSaveAs* function. Because the name of the drawing usually changes when the user calls the File | Save As command, you need to set the caption of the window to the file name. To do this, use the *SetCaption* function. This function takes a **char***, which in this case should be the *FileName* member of the *FileData* object. The *CmFileSaveAs* function should now look like this:

```
void
TDrawMDIChild::CmFileSaveAs()
{
  if (IsNewFile)
    strcpy(FileData->FileName, "");
  if ((TFileSaveDialog(this, *FileData)).Execute() == IDOK)
    SaveFile();
  SetCaption(FileData->FileName);
}
```

Creating the TDrawMDIChild constructor

The main difference between *TDrawMDIChild* and the *TDrawWindow* class, other than the fact that *TDrawMDIChild* has three fewer functions than *TDrawWindow*, is in the constructor.

Initializing data members

Like *TDrawWindow*, *TDrawMDIChild* contains the device context object that displays the drawing and manages the arrays that contain the line drawing information. It also contains the *IsDirty* flag, setting it to false when the drawing is first created or opened and setting it to true when the drawing is modified. So the variables that contain the data for these functions—*DragDC*, *Line*, *Lines*, and *IsDirty*—need to be initialized in the *TDrawMDIChild* constructor. This looks just the same as their initialization in the *TDrawWindow* class.

```
  DragDC = 0;
  Lines = new TLines(5, 0, 5);
  Line = new TLine(TColor::Black, 1);
  IsDirty = false;
```

There are some notable changes from *TDrawWindow*'s constructor here, however. First, the *Init* function is no longer called. *TMDIChild* does not provide an *Init* function. Instead, you should just call the base class constructor in the *TDrawMDIChild* initialization list, like so:

```
TDrawMDIChild::TDrawMDIChild(TDrawMDIClient& parent, const char* title)
  : TMDIChild(parent, title)
{
  :
}
```

Initializing file information data members

You can no longer simply initialize the *IsNewFile* variable to true, assuming that you are creating a new drawing whenever you create a window. In earlier steps this was a valid assumption: when the window was created, it hadn't opened a file yet, but was available to be drawn in. The *IsNewFile* flag was only set to false once a drawing had either been saved to a file or an existing drawing had been opened from a file into a window that had already been created.

In this case, the MDI client parent window will handle the file creation and opening operations. It then creates a child window to contain the new or existing drawing. The child window has to find out from the parent whether this is a new drawing or an existing drawing opened from a file.

For the same reason, the MDI child window does not necessarily create the *TOpenSaveDialog::TData* referenced by the *FileData* member. The *TDrawMDIClient* class has a function (or will have, when you get around to creating it) called *GetFileData*. This function takes no parameters and returns a pointer to a *TOpenSaveDialog::TData* object. If the MDI client window is creating the child window in response to a CM_FILEOPEN event, it creates a new *TOpenSaveDialog::TData* object containing the information about the file to be opened. *GetFileData* returns a pointer to that object. But if the client window is creating the child window in response to a CM_FILENEW event, *TDrawMDIClient* doesn't create a *TOpenSaveDialog::TData* object and *GetFileData* returns 0.

So the MDI child can find out whether this is a new drawing or not by testing the return value of *GetFileData*. If *GetFileData* returns a valid object, then it should assign the pointer to this object to its *FileData* member and set *IsNewFile* to false. It can then call the *OpenFile* function to load the drawing just as it did before. If *GetFileData* doesn't return a valid object (that is, it returns 0), the MDI child should set *IsNewFile* to true and create a new *TOpenSaveDialog::TData* object. The file name in the new object is set in the *CmFileSaveAs* function, just as it was in previous steps.

The constructor for *TDrawMDIChild* should look something like this:

```
TDrawMDIChild::TDrawMDIChild(TDrawMDIClient& parent, const char* title)
  : TMDIChild(parent, title)
{
  DragDC = 0;
```

```
    Lines = new TLines(5, 0, 5);
    Line = new TLine(TColor::Black, 1);
    IsDirty = false;

    // If the parent returns a valid FileData member, this is an open operation
    // Copy the parent's FileData member, since that'll go away
    if(FileData = parent.GetFileData()) {
      // Not a new file
      IsNewFile = false;
      OpenFile();
    }
    // But if the parent returns 0, this is a new operation
    else {
      // This is a new file
      IsNewFile = true;
      // Create a new FileData member
      FileData = new TOpenSaveDialog::TData(OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                                            "Point Files (*.PTS)|*.pts|", 0, "",
                                            "PTS");
    }
  }
```

Note that, in the case of an open operation, the child assigns the pointer returned by *GetFileData* to its *FileData* member. Once this is done, the child takes over responsibility for the *TOpenSaveDialog::TData* object, including responsibility for cleaning it up. Since this is already done in the destructor, you don't have to do anything else.

Creating the MDI client window class

The *TDrawMDIClient* class manages the multiple child windows open on its client area and all the attendant functionality, such as creating new children, closing windows either singly or all at one time, tiling or cascading the windows, and arranging the icons of minimized children. *TDrawMDIClient* inherits a great deal of this functionality from the *TMDIClient* class.

TMDIClient functionality

It is important to understand the *TMDIClient* class, for the main reason that it is going to do a lot of work for you. *TMDIClient* is virtually derived from the *TWindow* class. *TMDIClient* overrides two of *TWindow*'s virtual functions, *PreProcessMsg* and *Create*, to provide specific keyboard and menu handling functionality required by the client window. *TMDIClient* also handles a number of events, which are described in Table 11.2.

Table 11.2 Events handled by TMDIClient

Event	Response function	Purpose
CM_CREATECHILD	CmCreateChild	Creates a new MDI child window
CM_TILECHILDREN	CmTileChildren	Tiles all non-minimized MDI child windows vertically
CM_TILECHILDRENHORIZ	CmTileChildrenHoriz	Tiles all non-minimized MDI child windows horizontally

Table 11.2 Events handled by TMDIClient (continued)

Event	Response function	Purpose
CM_CASCADECHILDREN	CmCascadeChildren	Cascades all non-minimized MDI child windows
CM_ARRANGEICONS	CmArrangeIcons	Arranges the icons of all minimized MDI child windows
CM_CLOSECHILDREN	CmCloseChildren	Closes all open MDI child windows

The Drawing Pad application actually only provides menu items for four of these—CM_TILECHILDREN, CM_CASCADECHILDREN, CM_ARRANGEICONS, and CM_CLOSECHILDREN.

These response functions are simply wrappers for other *TMDIClient* functions that actually perform the work necessary. Each response function calls a function with the same name without the *Cm* prefix, so that *CmCreateChild* calls the *CreateChild* function. The only exception is *CmTileChildrenHoriz*, which calls the *TileChildren* function with the MDITILE_HORIZONTAL parameter.

Another function provided by *TMDIClient* is the *GetActiveMDIChild* function, which returns a pointer to the active MDI child window. Note that there can only be one active MDI child window at any time, but there is always one active MDI child window, even if all the MDI child windows are minimized.

There is one other function to discuss, *InitChild*. This is the only function in *TMDIClient* that you need to override in *TDrawMDIClient*. *InitChild* and overriding it to work with *TDrawMDIClient* are discussed on page 75.

Data members in TDrawMDIClient

TDrawMDIClient requires a couple of new data members. These should both be declared private.

The first is *NewChildNum*. The only function of this variable is to keep track of the number of new drawing created by the *CmFileNew* function. This number is used for the window caption of all new drawings. It is initialized to 0 in the *TDrawMDIClient* constructor.

The second is *FileData*, a pointer to a *TOpenSaveDialog::TData* object, just like the *FileData* member of *TDrawMDIChild*. *FileData* is used to hold the file information when a user opens an existing file. It is set to 0 in the constructor. *FileData* is also set to 0 once the MDI child window has been opened. As shown on page 71, the object returned by *GetFileData* is assigned to the *FileData* member of *TDrawMDIChild*. The object returned by *GetFileData* is actually the object (or lack thereof in the case of a new file) pointed to by *TDrawMDIClient*'s *FileData* member.

Adding response functions

In addition to the events handled by *TMDIClient*, *TDrawMDIClient* also handles the events formerly handled by *TDrawWindow* and not handled by *TDrawMDIChild*—CM_FILENEW, CM_FILEOPEN, and CM_ABOUT. The *CmAbout* response function is mostly unchanged from the *TDrawWindow* version, other than changing the class

specifier. On the other hand, the *CmFileNew* and *CmFileOpen* functions must be substantially changed.

CmFileNew

The *CmFileNew* function is actually simplified from its *TDrawWindow* version. It no longer has to deal with flushing the line arrays, invalidating the window, and setting flags. Instead it sets *FileData* to 0 so that the MDI child object can tell that it is displaying a new drawing, increments *NewChildNum*, then calls *CreateChild*. *CreateChild* is the function that actually creates and displays the new MDI child window. It is discussed in more detail in the discussion of the *InitChild* function on page 75.

The *CmFileNew* function should now look something like this:

```
void
TDrawMDIClient::CmFileNew()
{
  FileData = 0;
  NewChildNum++;
  CreateChild();
}
```

CmFileOpen

There are a number of differences between the *TDrawWindow* version of *CmFileOpen* and the *TDrawMDIClient* version.

- The *TDrawMDIClient* version no longer needs to call the *CanClose* function, because no windows need to be closed to open a new window.
- The *TDrawMDIClient* needs to create a new *TOpenSaveDialog::TData object to use with the TFileOpenDialog object.*
- If the call to *TFileOpenDialog.Execute* returns ID_OK, the *TDrawMDIClient* version calls *CreateChild* instead of *OpenFile.*
- Once the *CreateChild* call returns, you need to set *FileData* to 0. Although it may seem like you should delete the *FileData* object before discarding the pointer to it, the object is actually taken over by the MDI child object, which deletes the object when the MDI child is destroyed.

Your *CmFileOpen* function should look something like this:

```
void
TDrawMDIClient::CmFileOpen()
{
  // Create FileData.
  FileData = new TOpenSaveDialog::TData(OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                                        "Point Files (*.PTS)|*.pts|", 0, "",
                                        "PTS");
  // As long as the file open operation goes OK...
  if ((TFileOpenDialog(this, *FileData)).Execute() == IDOK)
    // Create the child window.
    CreateChild();
  // FileData is no longer needed.
```

```
  FileData = 0;
}
```

GetFileData

The only new function required for *TDrawMDIClient* is *GetFileData*. This function is called by *TDrawMDIChild* in its constructor. This function should take no parameters and return a pointer to a *TOpenSaveDialog::TData* object. Its function is to return a pointer to the object pointed to by *TDrawMDIClient*'s *FileData* member. If *FileData* references a valid object (that is, during a file open operation), *GetFileData* should return *FileData*. If *FileData* doesn't reference a valid object (that is, during a file new operation), *GetFileData* should return 0.

The actual function definition is very simple and can be inlined by defining the function inside the class declaration. Your *GetFileData* function should look something like this:

```
TOpenSaveDialog::TData *GetFileData() { return FileData ? FileData : 0; }
```

Overriding InitChild

The only *TMDIClient* function that *TDrawMDIChild* overrides is the *InitChild* function. *InitChild* takes no parameters and returns a pointer to a *TMDIChild* object. The *CreateChild* function calls *InitChild* before creating a new MDI child window. It is in *InitChild* that you create the *TMDIChild* or *TMDIChild*-derived object for the MDI child window. This is the only function of *TMDIClient* that you'll override when you create the *TDrawMDIClient* class.

The *InitChild* function for *TDrawMDIClient* is fairly straightforward. If *FileData* is 0, you should create a character array to contain a default window title. This can be initialized using the value of *NewChildNum* so that each new drawing has a different title.

Then you should create a *TMDIChild** and create a new *TDrawMDIChild* object. The constructor for *TDrawMDIChild* takes two parameters, a reference to a *TDrawMDIClient* object for its parent window and a **const char*** containing the MDI child window's caption. In this case, the first parameter should be the dereferenced **this** pointer. The second parameter should be either the *FileName* member of the *FileData* object if *FileData* references a valid object or the character array you created earlier if not.

Once the MDI child object has been created, you need to call the *SetIcon* function for the object. *SetIcon* associates an icon resource with the function's object. This icon is displayed in the client area when the child window is minimized. You can set the icon to the icon provided for the tutorial application called IDI_TUTORIAL.

The last step of the function is to return the *TMDIChild* pointer. Your *InitChild* function should look something like this:

```
TMDIChild*
TDrawMDIClient::InitChild()
{
  char title[15];
  if(!FileData)
    wsprintf(title, "New drawing %d", NewChildNum);
  TMDIChild* child = new TDrawMDIChild(*this, FileData ? FileData->FileName : title);
  child->SetIcon(GetApplication(), TResId("IDI_TUTORIAL"));
```

```
  return child;
}
```

Where to find more information

MDI frame, client, and child windows are described in Chapter 7 in the *ObjectWindows Programmer's Guide*.

Chapter

12

Using the Doc/View programming model

Step 12 introduces the Doc/View model of programming, which is based on the principle of separating data from the interface for that data. Essentially, the data is encapsulated in a document object, which is derived from the *TDocument* class, and displayed on the screen and manipulated by the user through a view object, which is derived from the *TView* class.

The Doc/View model permits a greater degree of flexibility in how you present data than does a model that links data encapsulation and user interface into a single class. Using the Doc/View model, you can define a document class to contain any type of data, such as a simple text file, a database file, or in this tutorial, a line drawing. You can then create a number of different view classes, each one of which displays the same data in a different manner or lets the user interact with that data in a different way.

For Step 12, however, you'll simply convert the application from its current model to the Doc/View model. Step 12 uses the SDI model so that you can more easily see the changes necessary for converting to Doc/View without being distracted by the extra code added in Step 11 to support MDI functionality. (You'll create an MDI Doc/View application in Step 13.) But even though the code for Step 12 will look very different from the code from Step 10, the running application for Step 12 will look nearly identical to that of Step 10. You can find the source for Step 12 in the files STEP12.CPP, STEP12.RC, STEP12DV.CPP, and STEP12DV.RC in the directory EXAMPLES\OWL\TUTORIAL.

Organizing the application source

The source for Step 12 is divided into four source files:

- STEP12.CPP contains the application object and its member definitions. It also contains the *OwlMain* function.

- STEP12.RC contains identifiers for events controlled by the application object, the resources for the frame window and its decorations, the About dialog box, and the application menu.
- STEP12DV.CPP contains the *TLine* class, the document class *TDrawDocument*, the view class *TDrawView*, and the associated member function definitions for each of these classes.
- STEP12DV.RC contains identifiers for events controlled by the view object and the resources for the view.

You should divide your Doc/View code this way to distinguish the document and its supporting view from the application code. The application code provides the support framework for the document and view classes, but doesn't contribute directly to the functionality of the Doc/View model. This also demonstrates good design practice for code reusability.

Doc/View model

The Doc/View model is based on three ObjectWindows classes:

- The *TDocument* class encapsulates and controls access to a set of data. A document object handles user access to that data through input from associated view objects. A document object can be associated with numerous views at the same time (for the sake of simplicity in this example, the document object is associated with only a single view object).
- The *TView* class provides an interface between a document object and the user interface. A view object controls how data from document object is displayed on the screen. A view object can be associated with only a single document object at any one time.
- The *TDocManager* class coordinates the associations between a document object and its view objects. The document manager provides a default File menu and default handling for each of the choices on the File menu. It also maintains a list of document templates, each of which specifies a relationship between a document class and a view class.

The *TDocument* and *TView* classes provide the abstract functionality for document and view objects. You must provide the specific functionality for your own document and view classes. You must also explicitly create the document manager and attach it to the application object. You must also provide the document templates for the document manager. These steps are described in the following sections.

TDrawDocument class

The *TDrawDocument* class is derived from the ObjectWindows class *TFileDocument*, which is in turn derived from the *TDocument* class. *TDocument* provides a number of input and output functions. These **virtual** functions return dummy values and have no

real functionality. *TFileDocument* provides the basic functionality required to access a data file in the form of a stream.

TDrawDocument uses the functionality contained in *TFileDocument* to access line data stored in a file. It uses a *TLines* array to contain the lines, the same as in earlier steps. The array is referenced through a pointer called *Lines*.

Creating and destroying TDrawDocument

TDrawDocument's constructor takes a single parameter, a *TDocument* *, that is a pointer to the parent document. A document can be a parent of a number of other documents, treating the data contained in those documents as if it were part of the parent. The constructor passes the parent pointer on to *TFileDocument*. The constructor also initializes the *Lines* data member to 0.

The destructor for *TDrawDocument* deletes the *TLines* object pointed to by *Lines*.

Storing line data

The document class you're going to create controls access to the data contained in a drawing. But you still need some way to store the data. You've already created the *TLine* class and the *TLines* array in previous steps. Luckily, this code can be recycled. The line data for each document is stored in a *TLines* array, and accessed by the document through a **protected** *TLines* * data member called *Lines*.

The *TPoints* and *TLines* arrays, their iterators, and the *TLine* class are now defined in the STEP12DV.CPP file. In the Doc/View model, these classes are an integral part of the document class you're about to build. The code for these classes doesn't change at all from Step 10.

Implementing TDocument virtual functions

TDrawDocument needs to implement a few of the **virtual** functions inherited from *TDocument*. These functions provide streaming and the ability to commit changes to the document or to discard all changes made to the document since the last save.

Opening and closing a drawing

Although *TFileDocument* provides the basic functionality required for stream input and output, it doesn't know how to read the data for a line. To provide this ability, you need to override the *Open* and *Close* functions.

Here's the signature of the *Open* function:

```
bool Open(int mode, const char far* path=0);
```

where:

- *mode* is the file open mode. In this case, you can ignore the mode parameter; the file is opened the same way each time, with the *ofRead* flag.

- *path* contains the document path. If a path is specified, the document's current path is changed to that path. If no path is specified (that is, *path* takes its default value), the path is left as it is. The path is used by the document when creating the document's streams.

The *Open* function is similar to the *OpenFile* function used in earlier steps in the tutorial. There are differences, though:

- The *Open* function creates the *TLines* array for the document object. In earlier steps, this was done in the *TDrawWindow* constructor, because *TDrawWindow* was responsible for containing all the *TLine* objects. Now the document is responsible for containing all the *TLine* objects, so it needs to create storage space for the data before it reads it in.
- If *path* is passed in, *Open* sets the document path to *path* with the *SetDocPath* function.
- *Open* checks whether the document has a path. If the document doesn't have a path, it is a new document, in which case there's no need to read in data from a file. If the document has a path, *Open* calls the *InStream* function. This function is defined in *TFileDocument* and returns a *TInStream* *.

  TInStream is the standard input stream class used by Doc/View classes. *TInStream* is derived from *TStream* and *istream*. *TStream* is an abstract base class that lets documents access standard streams. *TInStream* is essentially a standard *istream* adapted for use with the Doc/View model. There's also a corresponding *TOutStream* class, derived from *TStream* and *ostream*. You'll use *TOutStream* when you create the *Commit* function.
- After the input stream has been created, the data is read in and placed in the *TLines* array pointed to by *Lines*. When all the data is read in, the input stream is deleted.
- *Open* then calls the *SetDirty* function, passing false as the function parameter. The *SetDirty* function, and its equivalent access function *isDirty*, are the equivalent of the *IsDirty* flag in earlier steps of the tutorial. A document is considered to be dirty if it contains any changes to its data that have not been saved or committed.
- The last thing the *Open* function needs to do is return. If the document was successfully opened, *Open* returns true.

Here's how the code for your *Open* function might look:

```
bool
TDrawDocument::Open(int /*mode*/, const char far* path)
{
  Lines = new TLines(5, 0, 5);
  if (path)
    SetDocPath(path);
  if (GetDocPath()) {
    TInStream* is = InStream(ofRead);
    if (!is)
      return false;

    unsigned numLines;
    char fileinfo[100];
    *is >> numLines;
```

```
    is->getline(fileinfo, sizeof(fileinfo));
    while (numLines-) {
      TLine line;
      *is >> line;
      Lines->Add(line);
    }
    delete is;
  }
  SetDirty(false);
  NotifyViews(vnRevert, false);
  return true;
}
```

Closing the drawing is less complicated. The *Close* function discards the document's data and cleans up. In this case, it deletes the *TLines* array referenced by the *Lines* data member and returns true. Here's how the code for your *Close* function should look:

```
bool TDrawDocument::Close()
{
  delete Lines;
  Lines = 0;
  return true;
}
```

Lines is set to 0, both in the constructor and after closing the document, so that you can easily tell whether the document is open. If the document is open, *Lines* points to a *TLines* array, and is therefore not 0. But setting *Lines* to 0 makes it easy to check whether the document is open. The *IsOpen* function lets you check this from outside the document object:

```
bool IsOpen() { return Lines != 0; }
```

Saving and discarding changes

TDocument provides two functions for saving and discarding changes to a document:

- The *Commit* function commits changes made in the document's associated views by incorporating the changes into the document, then saving the data to persistent storage. *Commit* takes a single parameter, a bool. If this parameter is false, *Commit* saves the data only if the document is dirty. If the parameter is true, *Commit* does a complete write of the data. The default for this parameter is false.
- The *Revert* function discards any changes in the document's views, then forces the views to load the data contained in the document and display it. *Revert* takes a single parameter, a *bool*. If this parameter is true, the view clears its window and does not reload the data from the document. The default for this parameter is false.

For *TDrawDocument*, the document is updated as each line is drawn in the view window. The only function of *Commit* for the *TDrawDocument* class is to save the data to a file.

Commit checks to see if the document is dirty. If not, and if the force parameter is false, *Commit* returns true, indicating that the operation was successful.

If the document is dirty, or if the force parameter is true, *Commit* saves the data. The procedure to save the data is similar to the *SaveFile* function in previous steps, but, as with the *Open* function, there are a few differences.

Commit calls the *OutStream* function to open an output stream. This function is defined in *TFileDocument* and returns a *TOutStream* *. *Commit* then writes the data to the output stream. The procedure for this is almost exactly identical to that used in the old *SaveFile* function.

After writing the data to the output stream, *Commit* turns the *IsDirty* flag off by calling *SetDirty* with a false parameter. It then returns true, indicating that the operation was successful.

Here's how the code for your *Commit* function might look:

```
bool
TDrawDocument::Commit(bool force)
{
  if (!IsDirty() && !force)
    return true;

  TOutStream* os = OutStream(ofWrite);
  if (!os)
    return false;

  // Write the number of lines in the figure
  *os << Lines->GetItemsInContainer();

  // Append a description using a resource string
  *os << ' ' << string(*GetDocManager().GetApplication(),IDS_FILEINFO) << '

  // Get an iterator for the array of lines
  TLinesIterator i(*Lines);

  // While the iterator is valid (i.e. you haven't run out of lines)
  while (i) {
    // Copy the current line from the iterator and increment the array.
    *os << i++;
  }
  delete os;

  SetDirty(false);
  return true;
}
```

There's only one thing in the *Commit* function that you haven't seen before:

```
// Append a description using a resource string
*os << ' ' << string(*GetDocManager().GetApplication(), IDS_FILEINFO) << '
```

This uses a special constructor for the ANSI *string* class:

```
string(HINSTANCE instance, uint id, int len = 255);
```

This constructor lets you get a string resource from any Windows application. You specify the application by passing an HINSTANCE as the first parameter of the *string* constructor. In this case, you can get the current application's instance through the document manager. The *GetDocManager* function returns a pointer to the document's document manager. In turn, the *GetApplication* function returns a pointer to the application that contains the document manager. This is converted implicitly into an HINSTANCE by a conversion operator in the *TModule* class. The second parameter of the *string* constructor is the resource identifier of a string defined in STEP12DV.RC. This string contains version information that can be used to identify the application that created the document.

The *Revert* function takes a single parameter, a bool indicating whether the document's views need to refresh their display from the document's data. *Revert* calls the *TFileDocument* version of the *Revert* function, which in turn calls the *TDocument* version of *Revert*. The base class function calls the *NotifyViews* function with the ***vnRevert*** event. The second parameter of the *NotifyViews* function is set to the parameter passed to the *TDrawDocument::Revert* function. *TFileDocument::Revert* sets *IsDirty* to false and returns. If *TFileDocument::Revert* returns false, the *TDrawDocument* should also return false.

If *TFileDocument::Revert* returns true, the *TDrawDocument* function should check the parameter passed to *Revert*. If it is false (that is, if the view needs to be refreshed), *Revert* calls the *Open* function to open the document file, reload the data, and display it.

Here's how the code for your *Revert* function might look:

```
bool
TDrawDocument::Revert(bool clear)
{
  if (!TFileDocument::Revert(clear))
    return false;
  if (!clear)
    Open(0);
  return true;
}
```

Accessing the document's data

There are two main ways to access data in *TDrawDocument*: adding a line (such as a new line when the user draws in a view) and getting a reference to a line in the document (such as getting a reference to each line when repainting the window). You can add two functions, *AddLine* and *GetLine*, to take care of each of these actions.

The *AddLine* function adds a new line to the document's *TLines* array. The line is passed to the *AddLines* function as a *TLine* **&**. After adding the line to the array, *AddLine* sets the *IsDirty* flag to true by calling *SetDirty*. It then returns the index number of the line it just added. Here's how the code for your *AddLines* function might look:

```
int
TDrawDocument::AddLine(TLine& line)
{
  int index = Lines->GetItemsInContainer();
  Lines->Add(line);
```

```
    SetDirty(true);
    return index;
}
```

The *GetLine* function takes an **int** parameter. This **int** is the index of the desired line. *GetLine* should first check to see if the document is open. If not, it can try to open the document. If the document isn't open and *GetLine* can't open it, it returns 0, meaning that it couldn't find a valid document from which to get the line.

Once you know the document is valid, you should also check to make sure that the index isn't too high. Compare the index to the return value from the *GetItemsInContainer* function. As long as the index is less, you can return a pointer to the *TLine* object. Here's how the code for your *GetLine* function might look:

```
TLine*
TDrawDocument::GetLine(int index)
{
  if (!IsOpen() && !Open(ofRead | ofWrite))
    return 0;
  return index < Lines->GetItemsInContainer() ? &(*Lines)[index] : 0;
}
```

TDrawView class

The *TDrawView* class is derived from the ObjectWindows *TWindowView* class, which is in turn derived from the *TView* and *TWindow* classes. *TView* doesn't have any inherent windowing capabilities; a *TView*-derived class gets these capabilities by either adding a window member or pointer or by mixing in a window class with a view class.

TWindowView takes the latter approach, mixing *TWindow* and *TView* to provide a single class with both basic windowing and viewing capabilities. By deriving from this general-purpose class, *TDrawView* needs to add only the functionality required to work with the *TDrawDocument* class.

The *TDrawView* is similar to the *TDrawWindow* class used in previous steps. In fact, you'll see that a lot of the functions from *TDrawWindow* are brought directly to *TDrawView* with little or no modifications.

TDrawView data members

The *TDrawView* class has a number of **protected** data members.

```
TDC *DragDC;
TPen *Pen;
TLine *Line;
TDragDocument *DrawDoc;
```

Three of these should look familiar to you. *DragDC*, *Pen*, and *Line* perform the same function in *TDrawView* as they did in *TDrawWindow*.

Although a document can exist with no associated views, the opposite isn't true. A view must be associated with an existing document. *TDrawView* is attached to its document

when it is constructed. It keeps track of its document through a *TDrawDocument* * called *DrawDoc*. The base class *TView* has a *TDocument* * member called *Doc* that serves the same basic purpose. In fact, during base class construction, *Doc* is set to point at the *TDrawDocument* object passed to the *TDrawView* constructor. *DrawDoc* is added to force proper type compliance when the document pointer is accessed.

Creating the TDrawView class

The *TDrawView* constructor takes two parameters, a *TDrawDocument* & (a reference to the view's associated document) and a *TWindow* * (a pointer to the parent window). The parent window defaults to 0 if no value is supplied. The constructor passes its two parameters to the *TWindowView* constructor, and initializes the *DrawDoc* member to point at the document passed as the first parameter.

The constructor also sets *DragDC* to 0 and initializes *Line* with a new *TLine* object.

The last thing the constructor does is set up the view's menu. You can use the *TMenuDescr* class to set up a menu descriptor from a menu resource. Here's the *TMenuDescr* constructor:

```
TMenuDescr(TResId id);
```

where *id* is the resource identifier of the menu resource.

The *TMenuDescr* constructor takes the menu resource and divides it up into six groups. It determines which group a particular menu in the resource goes into by the presence of separators in the menu resource. The only separators that actually divide the resource into groups are at the pop-up level; that is, the separators aren't contained in a menu, but they're at the level of menu items that appear on the menu bar. For example, the following code shows a small snippet of a menu resource:

```
COMMANDS MENU
{
  // Always starts with the File group
  POPUP "&File"
  {
    MENUITEM "&Open", CM_FILEOPEN
    MENUITEM "&Save", CM_FILESAVE
  }
  MENUITEM SEPARATOR
    // Edit group
  MENUITEM SEPARATOR
    // Container group
  MENUITEM SEPARATOR
  // This one is in the Object group
  POPUP "&Objects"
  {
    MENUITEM "&Copy object", CM_OBJECTCOPY
    MENUITEM "Cu&t object", CM_OBJECTCUT
  }

  // No more items, meaning the Window group and Help group are also empty
}
```

A menu descriptor would separate this resource into groups like this: the File menu would be placed in the first group, called the File group. The second group (Edit group) and the third group (Container group) are empty, because there' s no pop-up menus between the separators that delimit those groups. The Tools menu is in the Object group. Because there are no menu resources after the Tools menu, the last two groups, the Object group and Help group, are also empty.

Although the groups have particular names, these names just represent a common name for the menu group. The menu represented by each group does not necessarily have that name. The document manager provides a default File menu, but the other menu names can be set in the menu resource.

In this case, the view supplies a menu resource called IDM_DRAWVIEW, which is contained in the file STEP12DV.RC. This menu is called Tools, which has the same choices on it as the Tools menu in earlier steps: Pen Size and Pen Color. To insert the Tools menu as the second menu on the menu bar when the view is created or activated, the menu resource is set up to place the Tools menu in the second group, the Edit group, so that the menu resource looks something like this:

```
IDM_DRAWVIEW MENU
{
  // Edit Group
  MENUITEM SEPARATOR
  POPUP "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }
}
```

You can install the menu descriptor as the view menu using the *TView* function *SetViewMenu* function, which takes a single parameter, a *TMenuDescr* *. *SetViewMenu* sets the menu descriptor as the view's menu. When the view is created, this menu is merged with the application menu.

Here's how the call to set up the view menu should look:

```
SetViewMenu(new TMenuDescr(IDM_DRAWVIEW));
```

The destructor for the view deletes the device context referenced by *DragDC* and the *TLine* object referenced by *Line*.

Naming the class

Every view class should define the function *StaticName*, which takes no parameters and returns a **static const char far** *. This function should return the name of the view class. Here's how the *StaticName* function might look:

```
static const char far* StaticName() {return "Draw View";}
```

Protected functions

TDrawView has a couple of **protected** access functions to provide functionality for the class.

The *GetPenSize* function is identical to the *TDrawWindow* function *GetPenSize*. This function opens a *TInputDialog*, gets a new pen size from the user, and changes the pen size for the window and calls the *SetPen* function of the current line.

The *Paint* function is a little different from the *Paint* function in the *TDrawWindow* class, but it does basically the same thing. Instead of using an iterator to go through the lines in an array, *TDrawView::Paint* calls the *GetLine* function of the view's associated document. The return from *GetLine* is assigned to a **const** *TLine* * called *line*. If *line* is not 0 (that is, if *GetLine* returned a valid line), *Paint* then calls the line's *Draw* function. Remember that the *TLine* class is unchanged from Step 10. The line draws itself in the window.

Here's how the code for the *Paint* function might look:

```
void
TDrawView::Paint(TDC& dc, bool, TRect&)
{
  // Iterates through the array of line objects.
  int i = 0;
  const TLine* line;
  while ((line = DrawDoc->GetLine(i++)) != 0)
    line->Draw(dc);
}
```

Event handling in TDrawView

The *TDrawView* class handles many of the events that were previously handled by the *TDrawWindow* class. Most of the other events that *TDrawWindow* handled that aren't handled by *TDrawView* are handled by the application object and the document manager; this is discussed later in Step 12.

In addition, *TDrawView* handles two new messages: VN_COMMIT and VN_REVERT. These view notification messages are sent by the view's document when the document's *Commit* and *Revert* functions are called.

Here's the response table definition for *TDrawView*:

```
DEFINE_RESPONSE_TABLE1(TDrawView, TWindowView)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
  EV_COMMAND(CM_PENSIZE, CmPenSize),
  EV_COMMAND(CM_PENCOLOR, CmPenColor),
  EV_VN_COMMIT,
  EV_VN_REVERT,
END_RESPONSE_TABLE;
```

The following functions are nearly the same in *TDrawView* as the corresponding functions in *TDrawWindow*. Any modifications to the functions are noted in the right column of the table:

Function	TDrawView version
EvLButtonDown	Does not set *IsDirty*. This is taken care of in *EvLButtonUp*.
EvRButtonDown	No change.
EvMouseMove	No change.
EvLButtonUp	Checks to see if the mouse was moved after the left button press. If so, calls the document's *AddLine* function to add the point.
CmPenSize	No change.
CmPenColor	No change.

The *VnCommit* function always returns true. In a more complex application, this function would add any cached data to the document, but in this application, the data is added to the document as each line is drawn.

The *VnRevert* function invalidates the display area, clearing it and repainting the drawing in the window. It then returns true.

Defining document templates

Once you've created a document class and an accompanying view class, you have to associate them so they can function together. An association between a document class and a view class is known as a document template class. The document template class is used by the document manager to determine what view class should be opened to display a document.

You can create a document template class using the macro DEFINE_DOC_TEMPLATE_CLASS, which takes three parameters. The first parameter is the name of the document class, the second is the name of the view class, and the third is the name of the document template class. The macro to create a template class for the *TDrawDocument* and *TDrawView* classes would look like this:

```
DEFINE_DOC_TEMPLATE_CLASS(TDrawDocument, TDrawView, DrawTemplate);
```

Once you've created a document template class, you need to create a document registration table. Document registration tables contain information about a particular Doc/View template class instance, such as what the template class does, the default file extension, and so on. A document registration table is actually an object of type *TRegList*, although you don't have to worry about what the object actually looks; you'll very rarely need to directly access a document registration table object.

Start creating a document registration table by declaring the BEGIN_REGISTRATION macro. This macro takes a single parameter, the name of the document registration class, which is used as the name of the *TRegList* object.

The next lines in your document registration table create entries in the document registration table. For a Doc/View template, you need to enter four items into this table:

- A description of the Doc/View template

- The default file extension when saving a file
- A filter string that is used to filter file names in the current directory
- Document creation flags

For the first three of these, you specify them using the REGDATA macro:

```
REGDATA(key, value)
```

key indicates what the *value* string pertains to. There are three different keys you need for creating a document registration table:

- *description* indicates *value* is the template description
- *extension* indicates value is the default file extension
- *docfilter* indicates *value* is the file-name filter
- The other macro you need to use to create a document registration table is the REGDOCFLAGS macro. This macro takes a single parameter, one or more document creation flags; if you specify more than one, the flags should be ORed together. For now, you can get by using two flags, *dtAutoDelete* and *dtHidden*. These flags are described in the *ObjectWindows Reference Guide* and Chapter 10 of the *ObjectWindows Programmer's Guide*.

A typical document registration table looks something like this:

```
BEGIN_REGISTRATION(DrawReg)
  REGDATA(description, "Point Files (*.PTS)")
  REGDATA(extension, ".PTS")
  REGDATA(docfilter, "*.pts")
  REGDOCFLAGS(dtAutoDelete | dtHidden)
END_REGISTRATION
```

Once you've created a document registration table, all you need to do is create an instance of the class. The class type is the name of the document template class. You also should give the instance a meaningful name. The constructor for any document template class looks like this:

```
TplName name(TRegList& reglist);
```

where:

- *TplName* is the class name you specified when defining the template class.
- *name* is whatever name you want to give this instance.
- *reglist* is the name of the registration table you created; it's the same name you passed as the parameter to the BEGIN_REGISTRATION macro.

Here's how the template instance for *TDrawDocument* and *TDrawView* classes might look:

```
DrawTemplate drawTpl(DrawReg);
```

Supporting Doc/View in the application

STEP12.CPP contains the code for the application object and the definition of the main window. The application object provides a framework for the Doc/View classes defined in STEP12DV.CPP. This section discusses the changes to the *TDrawApp* class that are required to support the new Doc/View classes. The *OwlMain* function remains unchanged.

InitMainWindow function

The *InitMainWindow* function requires some minor changes to support the Doc/View model:

- The *TDecoratedFrame* constructor takes a 0 in place of the *TDrawWindow* constructor for the frame's client window. The client window is set in the *EvNewView* function.
- The *AssignMenu* call is changed to a *SetMenuDescr* call. The *SetMenuDescr* function, which is inherited from *TFrameWindow*, takes a *TMenuDescr* as its only parameter. The *TMenuDescr* object should be built using the COMMANDS menu resource. This call looks something like this:

  ```
  GetMainWindow()->SetMenuDescr(TMenuDescr("COMMANDS"));
  ```

- A call to *SetDocManager* is added. This function sets the *DocManager* member of the *TApplication* class. It takes a single parameter, a *TDocManager* *.
- The *TDocManager* constructor takes a single parameter, which consists of one or more flags ORed together. The only flag that is required is either *dmSDI* or *dmMDI*. These flags set the document manager to supervise a single-document interface (*dmSDI*) or a multiple-document interface (*dmMDI*) application.

  In this case, you're creating an SDI application, so you should specify the *dmSDI* flag. In addition, you should specify the *dmMenu* flag, which instructs the document manager to provide its default menu.

  The call to the *SetDocManager* function should look like this:

  ```
  SetDocManager(new TDocManager(dmSDI | dmMenu));
  ```

InitInstance function

The *InitInstance* function is overridden because there are a couple of function calls that need to be made *after* the main window has been created. *InitInstance* should first call the *TApplication* version of *InitInstance*. That function calls the *InitMainWindow* function, which constructs the main window object, then creates the main window.

After the base class *InitInstance* function has been called, you need to call the main window's *DragAcceptFiles* function, specifying the true parameter. This enables the main window to accept files that are dropped in the window. Drag and drop functionality is handled through the application's response table, as discussed in the next section.

To enable the user to begin drawing in the window as soon as the application starts up, you also need to call the *CmFileNew* function of the document manager. This creates a new untitled document and view in the main window.

The *InitInstance* function should look something like this:

```
void
TDrawApp::InitInstance()
{
  TApplication::InitInstance();
  GetMainWindow()->DragAcceptFiles(true);
  GetDocManager()->CmFileNew();
}
```

Adding functions to TDrawApp

The *TDrawApp* class adds a number of new functions. It overrides the *TApplication* version of *InitInstance*. It adds a response table and takes the *CmAbout* function from the *TDrawWindow* class. It adds drag and drop capability by adding the EV_WM_DROPFILES macro to the response table and adding the *EvDropFiles* function to handle the event. It also handles a new event, WM_OWLVIEW, that indicates a view request message. Two functions handle this message. *EvNewView* handles a WM_OWLVIEW message with the *dnCreate* parameter. *EvCloseView* handles a WM_OWLVIEW message with the *dnClose* parameter.

Here's the new declaration of the *TDrawApp* class, along with its response table definition:

```
class TDrawApp : public TApplication
{
  public:
    TDrawApp() : TApplication() {}

  protected:
    // Override methods of TApplication
    void InitInstance();
    void InitMainWindow();

    // Event handlers
    void EvNewView  (TView& view);
    void EvCloseView(TView& view);
    void EvDropFiles(TDropInfo dropInfo);
    void CmAbout();
  DECLARE_RESPONSE_TABLE(TDrawApp);
};

DEFINE_RESPONSE_TABLE1(TDrawApp, TApplication)
  EV_OWLVIEW(dnCreate, EvNewView),
  EV_OWLVIEW(dnClose,  EvCloseView),
  EV_WM_DROPFILES,
  EV_COMMAND(CM_ABOUT, CmAbout),
END_RESPONSE_TABLE;
```

CmAbout function

The *CmAbout* function is nearly identical to the *TDrawWindow* version. The only difference is that the *CmAbout* function is no longer contained in its parent window class. Instead of using the **this** pointer as its parent, it substitutes a call to *GetMainWindow* function. The function should now look like this:

```
void
TDrawApp::CmAbout()
{
  TDialog(GetMainWindow(), IDD_ABOUT).Execute();
}
```

EvDropFiles function

The *EvDropFiles* function handles the WM_DROPFILES event. This function gets one parameter, a *TDropInfo* object. The *TDropInfo* object contains functions to find the number of files dropped, the names of the files, where the files were dropped, and so on.

Because this is a SDI application, if the number of files is greater than one, you need to warn the user that only one file can be dropped into the application at a time. To find the number of files dropped in, you can call the *TDropInfo* function *DragQueryFileCount*, which takes no parameters and returns the number of files dropped. If the file count is greater than one, pop up a message box to warn the user.

Now you need to get the name of the file dropped in. You can find the length of the file path string using the *TDropInfo* function *DragQueryFileNameLen*, which takes a single parameter, the index of the file about which you're inquiring. Because you know there's only one file, this parameter should be a 0. This function returns the length of the file path.

Allocate a string of the necessary length, then call the *TDropInfo* function *DragQueryFile*. This function takes three parameters. The first is the index of the file. Again, this parameter should be a 0. The second parameter is a **char** *, the file path. The third parameter is the length of the file path. This function fills in the file path in the **char** array from the second parameter.

Once you've got the file name, you need to get the proper template for the file type. To do this, call the document manager's *MatchTemplate* function. This function searches the document manager's list of document templates and returns a pointer to the first document template with a pattern that matches the dropped file. This pointer is a *TDocTemplate* *. If the document manager can't find a matching template, it returns 0.

Once you've located a template, you can call the template's *CreateDoc* function with the file path as the parameter to the function. This creates a new document and its corresponding view, and opens the file into the document.

Once the file has been opened, you must make sure to call the *DragFinish* function. This function releases the memory that Windows allocates during drag and drop operations.

Here's how the *EvDropFiles* function should look:

```
void
TDrawApp::EvDropFiles(TDropInfo dropInfo)
{
  if (dropInfo.DragQueryFileCount() != 1)
    ::MessageBox(0,"Can only drop 1 file in SDI mode","Drag/Drop Error",MB_OK);
  else {
    int fileLength = dropInfo.DragQueryFileNameLen(0)+1;
    char* filePath = new char [fileLength];
    dropInfo.DragQueryFile(0, filePath, fileLength);
    TDocTemplate* tpl = GetDocManager()->MatchTemplate(filePath);
    if (tpl)
      tpl->CreateDoc(filePath);
    delete filePath;
  }
  dropInfo.DragFinish();
}
```

EvNewView function

The WM_OWLVIEW event informs the application when a view-related event has happened. All functions that handle WM_OWLVIEW events return **void** and take a single parameter, a *TView* **&**. When the event's parameter is *dnCreate*, this indicates that a new view object has been created and requires the application to set up the view's window.

In this case, you need to set the view's window as the client of the main window. There are two functions you need to call to do this: *GetWindow* and *SetClientWindow*.

The *GetWindow* function is member of the view class. It takes no parameters and returns a *TWindow* *. This points to the view's window.

Once you have a pointer to the view's window, you can set that window as the client window with the main window's *SetClientWindow* function, which takes a single parameter, a *TWindow* *, and sets that window object as the client window. This function returns a *TWindow* *. This return value is a pointer to the old client window, if there was one.

Before continuing, you should check that the new client window was successfully created. *TView* provides the *IsOK* function, which returns false if the window wasn't created successfully. If *IsOK* returns false, you should call *SetClientWindow* again, passing a 0 as the window pointer, and return from the function.

If the window was created successfully, you need to check the view's menu with the *GetViewMenu* function. If the view has a menu, use the *MergeMenu* function of the main window to merge the view's menu with the window's menu.

The code for *EvNewView* should look like this:

```
void
TDrawApp::EvNewView(TView& view)
{
  GetMainWindow()->SetClientWindow(view.GetWindow());
  if (!view.IsOK())
```

```
    GetMainWindow()->SetClientWindow(0);
  else if (view.GetViewMenu())
    GetMainWindow()->MergeMenu(*view.GetViewMenu());
}
```

EvCloseView function

If the parameter for the WM_OWLVIEW event is *dnClose*, this indicates that a view has been closed. This is handled by the *EvCloseView* parameter. Like the *EvNewView* function, the *EvCloseView* function returns **void** and takes a *TView* **&** parameter.

To close a view, you need to remove the view's window as the client of the main window. To do this, call the main window's *SetClientWindow* function, passing a 0 as the window pointer. You can then restore the menu of the frame window to its former state using the *RestoreMenu* function of the main window.

When the *EvNewView* function creates a new view, the caption of the frame window is set to the file path of the document. You need to reset the main window's caption using the *SetCaption* function.

Here's the code for the *EvCloseView* function:

```
void
TDrawApp::EvCloseView(TView& /*view*/)
{
  GetMainWindow()>SetClientWindow(0);
  GetMainWindow()->RestoreMenu();
  GetMainWindow()->SetCaption("Drawing Pad");
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- The *InitMainWindow* and *InitInstance* functions are discussed in Chapter 2 in the *ObjectWindows Programmer's Guide*.
- Menu and menu descriptor objects are described in Chapter 8 in the *ObjectWindows Programmer's Guide*.
- The Doc/View classes are discussed in Chapter 10 in the *ObjectWindows Programmer's Guide*.
- The drag and drop functions are discussed in the *ObjectWindows Reference Guide*.

Chapter

13

Moving the Doc/View application to MDI

The Doc/View model is much more useful when it is used in a multiple-document interface (MDI) application. The ability to have multiple child windows in a frame lets you open more than one view for a document. You can find the source for Step 13 in the files STEP13.CPP, STEP13.RC, STEP13DV.CPP, and STEP13DV.RC in the directory EXAMPLES\OWL\TUTORIAL.

In Step 13, you'll add MDI capability to the application. This requires new functionality in the *TDrawDocument* and *TDrawView* classes. In addition, you'll add new features such as the ability to delete or modify an existing line and the ability to undo changes. You'll also create a new view class called *TDrawListView* to take advantage of the ability to display multiple views. *TDrawListView* shows an alternate view of the drawing stored in *TDrawDocument*, displaying it as a list of line information.

Supporting MDI in the application

STEP13.CPP contains the code for the application object and the definition of the main window. The application object provides a framework for the Doc/View classes defined in STEP13DV.CPP. This section discusses the changes to the *TDrawApp* class that are required to provide MDI support for your Doc/View application. The *OwlMain* function remains unchanged.

Changing to a decorated MDI frame

To support an MDI application, you need to change the *TDecoratedFrame* you've been using to a *TDecoratedMDIFrame*. Then, inside the decorated MDI frame, you need to create an MDI client window with the class *TMDIClient*. To easily locate the client window later, add a *TMDIClient* * to your *TDrawApp* class. Call the pointer *Client*. This client window contains the MDI child windows that display the various views.

The constructor for *TDecoratedMDIFrame* is described on page 66. The parameters for the constructor in this case are different from the parameters used in creating the decorated MDI frame used in Step 11.

- There's no menu resource for this window. Instead, you'll construct a *TMenuDescr*, just as you did for Step 12.
- You need to create the client window explicitly so that you can assign it to the *Client* data member. Unlike Step 11, where you used a custom client window class derived from *TMDIClient*, in this step you can use a *TMDIClient* object directly. The functionality that was added to the *TDrawMDIClient* class, such as opening files, creating new drawings, and so on, is now handled by the document manager. Thus, *TMDIClient* is sufficient to handle the chore of managing the MDI child windows.
- Lastly, you should turn menu tracking on.

The window constructor should look like this:

```
TDecoratedMDIFrame* frame = new TDecoratedMDIFrame("Drawing Pad", 0,
                                                   *(Client = new TMDIClient), true);
```

Changing the hint mode

You might have noticed in Step 12 that the hint text for control bar buttons didn't appear until you actually press the button. You can change the hint mode so that the text shows up when you just run the mouse over the top of the button.

To make this happen, call the control bar's *SetHintMode* function with the *TGadgetWindow::EnterHints* parameter:

```
cb->SetHintMode(TGadgetWindow::EnterHints);
```

This causes hints to be displayed when the cursor is over a button, even if the button isn't pressed. You can reset the hint mode by calling *SetHintMode* with the *TGadgetWindow::PressHints* parameter. You can also turn off menu tracking altogether by calling *SetHintMode* with the *TGadgetWindow::NoHints* parameter.

Setting the main window's menu

You need to change the *SetMenuDescr* call a little. The COMMANDS menu resource has been expanded to provide placeholder menus for the document manager's and views' menu descriptors. Also, the decorated MDI frame provides window management functions, such as cascading or tiling child windows, arranging the icons of minimized child windows, and so on.

The call to the *SetMenuDescr* function should now look like this:

```
GetMainWindow()->SetMenuDescr(TMenuDescr("COMMANDS"));
```

Setting the document manager

You also need to change how you create the document manager in an MDI application. The only change you need to make in this case is to change the *dmSDI* flag to *dmMDI*. You need to keep the *dmMenu* flag:

```
SetDocManager(new TDocManager(dmMDI | dmMenu));
```

InitInstance function

You need to make one change to the *InitInstance* function: remove the call to *CmFileNew*. This makes the frame open with no untitled documents. In the SDI application, opening the frame with an untitled document was OK. If the user opened a file, the untitled document was replaced by the new document. But in an MDI application, if the user opens an existing document, the untitled document remains open, requiring the user to close it before it'll go away.

Opening a new view

When you open a new view, you must provide a window for the view. In Step 12, *EvNewView* used the same client window again and again for every document and view. In an MDI application, you can open numerous windows in the *EvNewView* function. Each window you open inside the client area should be a *TMDIChild*. You can place your view inside the *TMDIChild* object by calling the view's *GetWindow* function for the child's client window.

Once you've created the *TMDIChild* object, you need to set its menu descriptor, but only if the view has a menu descriptor itself. After setting the menu descriptor, call the MDI child's *Create* function.

The *EvNewView* function should now look something like this:

```
void
TDrawApp::EvNewView(TView& view)
{
  TMDIChild* child = new TMDIChild(*Client, 0, view.GetWindow());
  if (view.GetViewMenu())
    child->SetMenuDescr(*view.GetViewMenu());
  child->Create();
}
```

Modifying drag and drop

In the SDI version of the tutorial application, you had to check to make sure the user didn't drop more than one file into the application area. But in MDI, if the user drops in more than one file, you can open them all, with each document in a separate window. Here's how to implement the ability to open multiple files dropped into your application:

- Find the number of files dropped into the application. Use the *DragQueryFileCount* function. Use a **for** loop to iterate through the files.

- For each file, get the length of its path and allocate a **char** array with enough room. Call the *DragQueryFile* function with the file's index (which you can track using the loop counter), the **char** array, and the length of the path.
- Once you've got the file name, you can call the document manager's *MatchTemplate* function to get the proper template for the file type. This is done the same way as in Step 12; see page 92.
- Once you've located a template, call the template's *CreateDoc* function with the file path as the parameter to the function. This creates a new document and its corresponding view, and opens the file into the document.
- Once all the files have been opened, call the *DragFinish* function. This function releases the memory that Windows allocates during drag and drop operations.

Here's how the new *EvDropFiles* function should look:

```
void
TDrawApp::EvDropFiles(TDropInfo dropInfo)
{
 int fileCount = dropInfo.DragQueryFileCount();
  for (int index = 0; index < fileCount; index++) {
    int fileLength = dropInfo.DragQueryFileNameLen(index)+1;
    char* filePath = new char [fileLength];
    dropInfo.DragQueryFile(index, filePath, fileLength);
    TDocTemplate* tpl = GetDocManager()->MatchTemplate(filePath);
    if (tpl)
      tpl->CreateDoc(filePath);
    delete filePath;
  }
  dropInfo.DragFinish();
}
```

Closing a view

In Step 12, when you wanted to close a view, you had to remove the view as a client window, restore the main window's menu, and reset the main window's caption. You no longer need to do any of this, because these tasks are handled by the MDI window classes. Here's how your *EvCloseView* function should look:

```
void
TDrawApp::EvCloseView(TView& /*view*/)
{  // nothing needs to be done here for MDI
}
```

Changes to TDrawDocument and TDrawView

You need to make the following changes in the *TDrawDocument* and *TDrawView* classes. These changes include defining new events, adding new event-handling functions, adding document property functions, and more.

Defining new events

First you need to define three new events to support the new features in the *TDrawDocument* and *TDrawView* classes. These view notification events are *vnDrawAppend*, *vnDrawDelete*, and *vnDrawModify*. These events should be **const int**s, and defined as offsets from the predefined value *vnCustomBase*. Using *vnCustomBase* ensures that your new events don't overlap any ObjectWindows events.

Next, use the NOTIFY_SIG macro to specify the signature of the event-handling function. The NOTIFY_SIG macro takes two parameters, the event name (such as *vnDrawAppend* or *vnDrawDelete*) and the parameter type to be passed to the event-handling function. The size of the parameter type can be no larger than a **long**; if the object being passed is larger than a **long**, you must pass it by pointer. In this case, the parameter is just an **unsigned int** to pass the index of the affected line to the event-handling function. The return value of the event-handling function is always **void**.

Lastly, you need to define the response table macro for each of these events. By convention, the macro name uses the event name, in all uppercase letters, preceded by EV_VN_. Use the **#define** macro to define the macro name. To define the macro itself, use the VN_DEFINE macro. Here's the syntax for the VN_DEFINE macro:

```
VN_DEFINE(eventName, functionName, paramSize)
```

where:

- *eventName* is the event name.
- *functionName* is the name of the event-handling function.
- *paramSize* is the size of the parameter passed to the event-handling function; this can have four different values:
  - void
  - int (size of an int parameter depends on the platform)
  - long (32-bit integer or far pointer)
  - pointer (size of a pointer parameter depends on the memory model)

  You should specify the value that most closely corresponds to the event-handling function's parameter type.

The full definition of the new events should look something like this:

```
const int vnDrawAppend = vnCustomBase+0;
const int vnDrawDelete = vnCustomBase+1;
const int vnDrawModify = vnCustomBase+2;

NOTIFY_SIG(vnDrawAppend, unsigned int)
NOTIFY_SIG(vnDrawDelete, unsigned int)
NOTIFY_SIG(vnDrawModify, unsigned int)

#define EV_VN_DRAWAPPEND VN_DEFINE(vnDrawAppend, VnAppend, int)
#define EV_VN_DRAWDELETE VN_DEFINE(vnDrawDelete, VnDelete, int)
#define EV_VN_DRAWMODIFY VN_DEFINE(vnDrawModify, VnModify, int)
```

Changes to TDrawDocument

TDrawDocument adds some new **protected** data members:

- *UndoLine* is a *TLine* *. It is used to store a line after the original in the *Lines* array is modified or deleted.
- *UndoState* is an **int**. It indicates the nature of the last user operation, so that an undo can be performed by reversing the operation. It can have one of four values:
  - *UndoNone* indicates that no operations have been performed to undo.
  - *UndoDelete* indicates that a line was deleted from the document.
  - *UndoAppend* indicates that a new line was added to the document.
  - *UndoModify* indicates that a line in the document was modified.
- *UndoIndex* is an **int**. It contains the index of the last modified line, so that the modification can be undone.
- *FileInfo* is a *string*. It contains information about the file. This string is equivalent to the file information stored in the *TDrawDocument::Commit* function of Step 12.

The *TDrawDocument* constructor should be modified to initialize *UndoLine* to 0 and *UndoState* to *UndoNone*. The *TDrawDocument* destructor is modified to delete *UndoLine*.

You need to modify the *Open* function slightly to read the file information string from the document file and use it to initialize the *FileInfo* member. If the document doesn't have a valid document path, initialize *FileInfo* using the string resource IDS_FILEINFO.

Modify the *AddLine* function to notify any other views when a line has been added to the drawing. You can use the *NotifyViews* function with the *vnDrawAppend* event. The second parameter to the *NotifyViews* call should be the new line's array index. You also need to set *UndoState* to *UndoAppend*. The *AddLine* function should now look like this:

```
int
TDrawDocument::AddLine(TLine& line)
{
  int index = Lines->GetItemsInContainer();
  Lines->Add(line);
  SetDirty(true);
  NotifyViews(vnDrawAppend, index);
  UndoState = UndoAppend;
  return index;
}
```

Property functions

Every document has a list of properties. Each property has an associated value, defined as an **enum**, by which it is identified. The list of **enum**s for a derived document object should always end with the value *NextProperty*. The list of **enum**s for a derived document object should always start with the value *PrevProperty*, which should be set to the *NextProperty* member of the base class, minus 1.

Each property also has a text string describing the property contained in an array called *PropNames* and an **int** containing implementation-defined flags in an array called

PropFlags. The property's **enum** value can be used in an array index to locate the property string or flag for a particular property.

TDrawDocument adds two new properties to its document properties list: *LineCount* and *Description*. The **enum** definition should look like this:

```
enum {
  PrevProperty = TFileDocument::NextProperty-1,
  LineCount,
  Description,
  NextProperty,
};
```

By redefining *PrevProperty* and *NextProperty*, any class that's derived from your document class can create new properties without overwriting the properties you've defined.

TDrawDocument also adds an array of **static char** strings. This array contains two strings, each containing a text description of one of the new properties. The array definition should look like this:

```
static char* PropNames[] = {
  "Line Count",
  "Description",
};
```

Lastly, *TDrawDocument* adds an array of **ints** called *PropFlags*, which contains the same number of array elements as *PropNames*. Each array element contains one or more document property flags ORed together, and corresponds to the property in *PropNames* with the same array index. The *PropFlags* array definition should look like this:

```
static int PropFlags[] = {
  pfGetBinary|pfGetText, // LineCount
  pfGetText,             // Description
};
```

TDrawDocument overrides a number of the *TDocument* property functions to provide access to the new properties. You can find the total number of properties for the *TDrawDocument* class by calling the *PropertyCount* function. *PropertyCount* returns the value of the property **enum** *NextProperty*, minus 1.

You can find the text name of any document property using the *PropertyName* function. *PropertyName* returns a **char ***, a string containing the property name. It takes a single **int** parameter, which indicates the index of the parameter for which you want the name. If the index is less than or equal to the **enum** *PrevProperty*, you can call the *TFileDocument* function *PropertyName*. This returns the name of a property defined in *TFileDocument* or its base class *TDocument*. If the index is greater than or equal to *NextProperty*, you should return 0; *NextProperty* marks the last property in the document class. If the index has the same or greater value than *NextProperty*, the index is too high to be valid. As long as the index is greater than *PrevProperty* but less than *NextProperty*, you should return the string from the *PropNames* array corresponding to the index. The code for this function should look like this:

```
const char*
TDrawDocument::PropertyName(int index)
```

```
{
  if (index <= PrevProperty)
    return TFileDocument::PropertyName(index);
  else if (index < NextProperty)
    return PropNames[index-PrevProperty-1];
  else
    return 0;
}
```

The *FindProperty* function is essentially the opposite of the *PropertyName* function. *FindProperty* takes a single parameter, a **const char** *. It tries to match the string passed in with the name of each document property. If it successfully matches the string with a property name, it returns an **int** containing the index of the property. The code for this function should look like this:

```
int
TDrawDocument::FindProperty(const char far* name)
{
  for (int i=0; i < NextProperty-PrevProperty-1; i++)
    if (strcmp(PropNames[i], name) == 0)
      return i+PrevProperty+1;
  return 0;
}
```

The *PropertyFlags* function takes a single **int** parameter, which indicates the index of the parameter for which you want the property flags. These flags are returned as an **int**. If the index is less than or equal to the **enum** *PrevProperty*, you can call the *TFileDocument* function *PropertyName*. This returns the name of a property defined in *TFileDocument* or its base class *TDocument*. If the index is greater than or equal to *NextProperty*, you should return 0; *NextProperty* marks the last property in the document class. If the index has the same or greater value than *NextProperty*, the index is too high to be valid. As long as the index is greater than *PrevProperty* but less than *NextProperty*, you should return the member of the *PropFlags* array corresponding to the index. The code for this function should look like this:

```
int
TDrawDocument::PropertyFlags(int index)
{
  if (index <= PrevProperty)
    return TFileDocument::PropertyFlags(index);
  else if (index < NextProperty)
    return PropFlags[index-PrevProperty-1];
  else
    return 0;
}
```

The last property function is the *GetProperty* function, which takes three parameters. The first parameter is an **int**, the index of the property you want. The second parameter is a **void** *. This should be a block of memory that is used to hold the property information. The third parameter is an **int** and indicates the size in bytes of the block of memory.

There are three possibilities the *GetProperty* function should handle:

- The *LineCount* property can be requested in two forms, text or binary. To get the *LineCount* property in binary form, call the *GetProperty* function with the third parameter set to 0. If you do this, the second parameter should point to a data object of the proper type to contain the property data. To get the *LineCount* property as text, call the *GetProperty* function with the second parameter pointing to a valid block of memory and the third parameter set to the size of that block.
- The *Description* property can be requested in text form only. Just copy the *FileInfo* string into the destination array passed in as the second parameter.
- If the property requested is neither *LineCount* nor *Description*, call the *TFileDocument* version of *GetProperty*.

The code for the *GetProperty* function should look like this:

```
int
TDrawDocument::GetProperty(int prop, void far* dest, int textlen)
{
  switch(prop)
  {
    case LineCount:
    {
      int count = Lines->GetItemsInContainer();
      if (!textlen) {
        *(int far*)dest = count;
        return sizeof(int);
      }
      return wsprintf((char far*)dest, "%d", count);
    }
    case Description:
      char* temp = new char[textlen]; // need local copy for medium model
      int len = FileInfo.copy(temp, textlen);
      strcpy((char far*)dest, temp);
      return len;
  }
  return TFileDocument::GetProperty(prop, dest, textlen);
}
```

New functions in TDrawDocument

Step 13 adds a number of new functions to *TDrawDocument*. These functions let you modify the document object by deleting lines, modifying lines, clearing the document, and undoing changes.

The first new function is *DeleteLine*. As its name implies, the purpose of this function is to delete a line from the document. *DeleteLine* takes a single **int** parameter, which gives the array index of the line to be deleted.

- *Delete* should check that the index passed in to it is valid. You can check this by calling the *GetLine* function and passing the index to *GetLine*. If the index is valid, *GetLine* returns a pointer to a line object. Otherwise, it returns 0.

- Once you have determined the index is valid, you should set *UndoLine* to the line to be deleted and set *UndoState* to *UndoDelete*. This saves the old line in case the user requests an undo of the deletion.
- You should then detach the line from the document using the container class *Detach* function. This function takes a single **int** parameter, the array index of the line to be deleted.
- Turn the *IsDirty* flag on by calling the *SetDirty* function.
- Lastly, notify the views that the document has changed by calling the *NotifyViews* function. Pass the *vnDrawDelete* event as the first parameter of the *NotifyViews* call and the array index of the line as the second parameter.

The code for the *DeleteLine* function should look like this:

```
void
TDrawDocument::DeleteLine(unsigned int index)
{
  const TLine* oldLine = GetLine(index);
  if (!oldLine)
    return;
  delete UndoLine;
  UndoLine = new TLine(*oldLine);
  Lines->Detach(index);
  SetDirty(true);
  NotifyViews(vnDrawDelete, index);
  UndoState = UndoDelete;
}
```

The *ModifyLine* function takes two parameters, a *TLine* **&** and an **int**. The **int** is the array index of the line to be modified. The affected line is replaced by the *TLine* **&**.

- As with the *DeleteLine* function, you need to set up the undo data members before replacing the line. Copy the line to be replaced to *UndoLine* and set *UndoState* to *UndoModify*. You also need to set *UndoIndex* to the index of the affected line.
- Set the line to the *TLine* object passed into the function.
- Turn the *IsDirty* flag on by calling the *SetDirty* function.
- Lastly, notify the views that the document has changed by calling the *NotifyViews* function. Pass the *vnDrawModify* event as the first parameter of the *NotifyViews* call and the array index of the line as the second parameter.

The code for this function should look like this:

```
void
TDrawDocument::ModifyLine(TLine& line, unsigned int index)
{
  delete UndoLine;
  UndoLine = new TLine((*Lines)[index]);
  SetDirty(true);
  (*Lines)[index] = line;
  NotifyViews(vnDrawModify, index);
  UndoState = UndoModify;
```

```
  UndoIndex = index;
}
```

The *Clear* function is fairly straightforward. It flushes the *TLines* array referenced by *Lines*, then forces the views to update by calling *NotifyViews* with the *vnRevert* parameter. When the views are updated, there's no data in the document, causing the views to clear their windows. The function should look something like this:

```
void
TDrawDocument::Clear()
{
  Lines->Flush();
  NotifyViews(vnRevert, true);
}
```

The *Undo* function has three different types of operations to undo: append, delete, and modify. It determines which type of operation it needs to undo by the value of the *UndoState* variable:

- If *UndoState* is *UndoAppend*, *Undo* needs to delete the last line in the array.
- If *UndoState* is *UndoDelete*, *Undo* needs to add the line referenced by *UndoLine* to the array.
- If *UndoState* is *UndoModify*, *Undo* needs to restore the line referenced by *UndoLine* to the array to the position in the array indicated by *UndoIndex*.

Here's how the code for the *Undo* function should look:

```
void
TDrawDocument::Undo()
{
  switch (UndoState) {
    case UndoAppend:
      DeleteLine(Lines->GetItemsInContainer()-1);
      return;
    case UndoDelete:
      AddLine(*UndoLine);
      delete UndoLine;
      UndoLine = 0;
      return;
    case UndoModify:
      TLine* temp = UndoLine;
      UndoLine = 0;
      ModifyLine(*temp, UndoIndex);
      delete temp;
  }
}
```

Each operation uses one of these new modification functions. That way, each undo operation can itself be undone.

Changes to TDrawView

TDrawView modifies a number of its functions, including deleting the *GetPenSize* function. This function should be moved to the *TLine* class, so that the pen size is set in the line itself. You can call the *TLine::GetPenSize* function from the *CmPenSize* function. The same thing should be done with the *CmPenColor* function; move the functionality of this function to the *TLine::GetPenColor* function. You can call the *TLine::GetPenColor* function from the *CmPenColor* function.

To accommodate the new editing functionality in the *TDrawDocument* and *TDrawView* classes, you need to add menu choices for Undo and Clear. These choices should post the events CM_CLEAR and CM_UNDO. The menu requires a change in the menu resource to group the menus properly. The call should look like this:

```
SetViewMenu(new TMenuDescr(IDM_DRAWVIEW));
```

You can redefine the right button behavior by changing the *EvRButtonDown* function (there are now two other ways to change the pen size, the Tools | Pen Size menu command and the Pen Size control bar button). You can use the right mouse button as a shortcut for an undo operation. The *EvRButtonDown* function should look like this:

```
void
TDrawView::EvRButtonDown(uint, TPoint&)
{
  CmUndo();
}
```

New functions in TDrawView

Step 13 adds a number of new functions to *TDrawDocument*. These functions implement an interface to access the new functionality in *TDrawDocument*.

You need to override the *TView* **virtual** function *GetViewName*. The document manager calls this function to determine the type of view. This function should return a **const char *** referencing a string containing the view name. This function should look like this:

```
const char far* GetViewName() { return StaticName(); }
```

After adding the new menu items Clear and Undo to the Edit menu, you need to handle the events CM_CLEAR and CM_UNDO. Add the following lines to your response table:

```
EV_COMMAND(CM_CLEAR, CmClear),
EV_COMMAND(CM_UNDO, CmUndo),
```

You also need functions to handle the CM_CLEAR and CM_UNDO events. If the view receives a CM_CLEAR message, all it needs to do is to call the document's *Clear* function:

```
void
TDrawView::CmClear()
{
  DrawDoc->Clear();
}
```

If the view receives a CM_UNDO message, all it needs to do is to call the document's *Undo* function:

```
void
TDrawView::CmUndo()
{
  DrawDoc->Undo();
}
```

The other new events the view has to handle are the view notification events, *vnDrawAppend*, *vnDrawDelete*, and *vnDrawModify*. You should add the response table macros for these events to the view's response table:

```
DEFINE_RESPONSE_TABLE1(TDrawView, TWindowView)
  EV_VN_DRAWAPPEND,
  EV_VN_DRAWDELETE,
  EV_VN_DRAWMODIFY,
END_RESPONSE_TABLE;
```

The event-handling functions for these macros are *VnAppend*, *VnDelete*, and *VnModify*. All three of these functions return a bool and take a single parameter, an **int** indicating which line in the document is affected by the event.

The *VnAppend* function gets notification that a line was appended to the document. It then draws the new line in the view's window. It should create a device context, get the line from the document, call the line's *Draw* function with the device context object as the parameter, then return true. The code for this function looks like this:

```
bool
TDrawView::VnAppend(unsigned int index)
{
  TClientDC dc(*this);
  const TLine* line = DrawDoc->GetLine(index);
  line->Draw(dc);
  return true;
}
```

The *VnModify* function forces a repaint of the entire window. It might seem more efficient to just redraw the affected line, but you would need to paint over the old line, repaint the new line, and restore any lines that might have crossed or overlapped the affected line. It is actually more efficient to invalidate and repaint the entire window. So the code for the *VnModify* function should look like this:

```
bool
TDrawView::VnModify(unsigned int /*index*/)
{
  Invalidate();  // force full repaint
  return true;
}
```

The *VnDelete* function also forces a repaint of the entire window. This function faces the same problem as *VnModify*; simply erasing the line will probably affect other lines. The code for the *VnDelete* function should look like this:

```
bool
TDrawView::VnDelete(unsigned int /*index*/)
{
  Invalidate();  // force full repaint
  return true;
}
```

TDrawListView

The purpose of the *TDrawListView* class is to display the data contained in a *TDrawDocument* object as a list of lines. Each line will display the color values for the line, the pen size for the line, and the number of points that make up the line. *TDrawListView* will let the user modify a line by changing the pen size or color. The user can also delete a line.

TDrawListView is derived from *TView* and *TListBox*. *TView* gives *TDrawListView* the standard view capabilities. *TListBox* provides the ability to display the information in the document object in a list.

Creating the TDrawListView class

The *TDrawListView* constructor takes two parameters, a *TDrawDocument* & (a reference to the view's associated document) and a *TWindow* * (a pointer to the parent window). The parent window defaults to 0 if no value is supplied. The constructor passes the first parameter to the *TView* constructor and initializes the *DrawDoc* member to point at the document passed as the first parameter.

TDrawListView has two data members, one **protected** *TDrawDocument* * called *DrawDoc* and one **public int** called *CurIndex*. *DrawDoc* serves the same purpose in *TDrawListView* as it did in *TDrawView*, namely to reference the view's associated document object. *CurIndex* contains the array index of the currently selected line in the list box.

The *TDrawListView* constructor also calls the *TListBox* constructor. The first parameter of the *TListBox* constructor is passed the parent window parameter of the *TDrawListView* constructor. The second parameter of the *TListBox* constructor is a call to the *TView* function *GetNextViewId*. This function returns a **static unsigned** that is used as the list box identifier. The view identifier is set in the *TView* constructor. The coordinates and dimensions of the list box are all set to 0; the dimensions are filled in when the *TDrawListView* is set as a client in an MDI child window.

The constructor also sets some window attributes, including the *Attr.Style* attribute, which has the WS_BORDER and LBS_SORT attributes turned off, and the *Attr.AccelTable* attribute, which is set to the IDA_DRAWLISTVIEW accelerator resource defined in STEP13DV.RC.

The constructor also sets up the menu descriptor for *TDrawListView*. Because *TDrawListView* has a different function from *TDrawView*, it requires a different menu. Compare the menu resource for *TDrawView* and the menu resource for *TDrawListView*.

Here's the code for the *TDrawListView* constructor:

```
TDrawListView::TDrawListView(TDrawDocument& doc,TWindow *parent)
  : TView(doc), TListBox(parent, GetNextViewId(), 0,0,0,0), DrawDoc(&doc)
{
  Attr.Style &= ~(WS_BORDER | LBS_SORT);
  Attr.AccelTable = IDA_DRAWLISTVIEW;
  SetViewMenu(new TMenuDescr(IDM_DRAWLISTVIEW));
}
```

TDrawListView has no dynamically allocated data members. The destructor therefore does nothing.

Naming the class

Like the *TDrawView* class, *TDrawListView* should define the function *StaticName* to return the name of the view class. Here's how the *StaticName* function might look:

```
static const char far* StaticName() {return "DrawList View";}
```

Overriding TView and TWindow virtual functions

The document manager calls the view function *GetViewName* to determine the type of view. You need to override this function, which is declared **virtual** function in *TView*. This function should return a **const char** * referencing a string containing the view name. This function should look like this:

```
const char far* GetViewName() { return StaticName(); }
```

The document manager calls the view function *GetWindow* to get the window associated with a view. You need to override this function also, which is declared **virtual** function in *TView*. It should return a *TWindow* * referencing the view's window. This function should look like this:

```
TWindow* GetWindow() { return (TWindow*) this; }
```

You also need to supply a version of the *CanClose* function. This function should call the *TListBox* version of *CanClose* and also call the document's *CanClose* function. This function should look like this:

```
bool CanClose() {return TListBox::CanClose() && Doc->CanClose();}
```

You also need to provide a version of the *Create* function. You can call the *TListBox* version of *Create* to actually create the window. But you also need to load the data from the document into the *TDrawListView* object. To do this, call the *LoadData* function. You'll define the *LoadData* function in the next section of this step. The *Create* function should look something like this:

```
bool
TDrawListView::Create()
{
  TListBox::Create();
  LoadData();
  return true;
}
```

Loading and formatting data

You need to provide functions to load data from the document object to the view document and to format the data for display in the list box. These functions should be **protected** so that only the view can call them.

The first function is *LoadData*. To load data into the list box, you need to first clear the list of any items that might already be in it. For this, you can call the *ClearList* function, which is from the *TListBox* base class. After that, get lines from the document and format each line until the document runs out of lines. You can tell when there are no more lines in the document; the *GetLine* function returns 0. Lastly, set the current selection index to 0 using the *SetSelIndex* function. This causes the first line in the list box to be selected. The code for the *LoadData* function looks something like this:

```
void
TDrawListView::LoadData()
{
  ClearList();
  int i = 0;
  const TLine* line;
  while ((line = DrawDoc->GetLine(i)) != 0)
    FormatData(line, i++);
  SetSelIndex(0);
}
```

The *FormatData* function takes two parameters. The first parameter is a **const** *TLine* * that references the line to modified or added to the list box. The second parameter contains the index of the line to modified.

The code for *FormatData* should look something like this:

```
void
TDrawListView::FormatData(const TLine* line, int unsigned index)
{
  char buf[80];
  TColor color(line->QueryColor());
  wsprintf(buf, "Color = R%d G%d B%d, Size = %d, Points = %d",
           color.Red(), color.Green(), color.Blue(),
           line->QueryPenSize(), line->GetItemsInContainer());

  DeleteString(index);
  InsertString(buf, index);
  SetSelIndex(index);
}
```

Event handling in TDrawListView

Here's the response table for *TDrawListView*:

```
DEFINE_RESPONSE_TABLE1(TDrawListView, TListBox)
  EV_COMMAND(CM_PENSIZE, CmPenSize),
  EV_COMMAND(CM_PENCOLOR, CmPenColor),
  EV_COMMAND(CM_CLEAR, CmClear),
```

```
  EV_COMMAND(CM_UNDO, CmUndo),
  EV_COMMAND(CM_DELETE, CmDelete),
  EV_VN_ISWINDOW,
  EV_VN_COMMIT,
  EV_VN_REVERT,
  EV_VN_DRAWAPPEND,
  EV_VN_DRAWDELETE,
  EV_VN_DRAWMODIFY,
END_RESPONSE_TABLE;
```

This response table is similar to *TDrawView*'s response table in some ways. The two views share some events, such as the CM_PENSIZE and CM_PENCOLOR events and the *vnDrawAppend* and *vnDrawModify* view notification events.

But each view also handles events that the other view doesn't. This is because each view has different capabilities. For example, the *TDrawView* class handles a number of mouse events, whereas *TDrawListView* handles none. That's because it makes no sense in the context of a list box to handle the mouse events; those events are used when drawing a line in the *TDrawView* window.

TDrawListView handles the CM_DELETE event, whereas *TDrawView* doesn't. This is because, in the *TDrawView* window, there's no way for the user to indicate which line should be deleted. But in the list box, it's easy: just delete the line that's currently selected in the list box.

TDrawListView also handles the *vnIsWindow* event. The *vnIsWindow* message is a predefined ObjectWindows event, which asks the view if its window is the same as the window passed with the event.

The *CmPenSize* function is more complicated in the *TDrawListView* class than in the *TDrawView* class. This is because the *TDrawListView* class doesn't maintain a pointer to the current line the way *TDrawView* does. Instead, you have to get the index of the line that's currently selected in the list box and get that line from the document. Then, because the *GetLine* function returns a pointer to a **const** object, you have to make a copy of the line, modify the copy, then call the document's *ModifyLine* function. Here's how the code for this function should look:

```
void
TDrawListView::CmPenSize()
{
  int index = GetSelIndex();
  const TLine* line = DrawDoc->GetLine(index);
  if (line) {
    TLine* newline = new TLine(*line);
    if (newline->GetPenSize())
      DrawDoc->ModifyLine(*newline, index);
    delete newline;
  }
}
```

The interesting aspect of this function comes in the *ModifyLine* call. When the user changes the pen size using this function, the pen size in the view isn't changed at this time. But when the document changes the line in the *ModifyLine* call, it posts a *vnDrawModify* event to all of its views:

```
NotifyViews(vnDrawModify, index);
```

This notifies all the views associated with the document that a line has changed. All views then call their *VnModify* function and update their displays from the document. This way, any change made in one view is automatically reflected in other open views. The same holds true for any other functions that modify the document's data, such as *CmPenColor*, *CmDelete*, *CmUndo*, and so on.

The *CmPenColor* function looks nearly same as the *CmPenSize* function, except that, instead of calling the line's *GetPenSize* function, it calls *GetPenColor*:

```
void
TDrawListView::CmPenColor()
{
  int index = GetSelIndex();
  const TLine* line = DrawDoc->GetLine(index);
  if (line) {
    TLine* newline = new TLine(*line);
    if (newline->GetPenColor())
      DrawDoc->ModifyLine(*newline, index);
    delete newline;
  }
}
```

The CM_DELETE event indicates that the user wants to delete the line that is currently selected in the list box. The view needs to call the document's *DeleteLine* function, passing it the index of the currently selected line. This function should look like this:

```
void
TDrawListView::CmDelete()
{
  DrawDoc->DeleteLine(GetSelIndex());
}
```

You also need functions to handle the CM_CLEAR and CM_UNDO events for *TDrawListView*. If the user chooses the Clear menu command, the view receives a CM_CLEAR message. All it needs to do is call the document's *Clear* function:

```
void
TDrawListView::CmClear() {
  DrawDoc->Clear();
}
```

If the user chooses the Clear menu command, the view receives a CM_UNDO message. All it needs to do is call the document's *Undo* function:

```
void
TDrawListView::CmUndo()
{
  DrawDoc->Undo();
}
```

These functions are identical to the *TDrawView* versions of the same functions. That's because these operation rely on *TDrawDocument* to actually make the changes to the data.

Like the *TDrawView* class, *TDrawListView*'s *VnCommit* function always returns true. In a more complex application, this function would add any cached data to the document, but in this application, the data is added to the document as each line is drawn.

The *VnRevert* function calls the *LoadData* function to revert the list box display to the data contained in the document:

```
bool
TDrawListView::VnRevert(bool /*clear*/)
{
  LoadData();
  return true;
}
```

The *VnAppend* function gets a single **unsigned int** parameter, which gives the index number of the appended line. You need to get the new line from the document by calling the document's *GetLine* function. Call the *FormatData* function with the line and the line index passed into the function. After formatting the line, set the selection index to the new line and return. The function should look like this:

```
bool
TDrawListView::VnAppend(unsigned int index)
{
  const TLine* line = DrawDoc->GetLine(index);
  FormatData(line, index);
  SetSelIndex(index);
  return true;
}
```

The *VnDelete* function takes a single **int** parameter, the index of the line to be deleted. To remove the line from the list box, call the *TListBox* function *DeleteString*:

```
bool
TDrawListView::VnDelete(unsigned int index)
{
  DeleteString(index);
  HandleMessage(WM_KEYDOWN,VK_DOWN); // force selection
  return true;
}
```

The call to *HandleMessage* ensures that there is an active selection in the list box after the currently selected string is deleted.

The *VnModify* function takes a single **int** parameter, the index of the line to be modified. You need to get the line from the document using the *GetLine* function. Call *FormatData* with the line and its index:

```
bool
TDrawListView::VnModify(unsigned int index)
{
  const TLine* line = DrawDoc->GetLine(index);
  FormatData(line, index);
  return true;
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- The MDI window classes are discussed in Chapter 7 in the *ObjectWindows Programmer's Guide*.
- Menu descriptors are discussed in Chapter 8 in the *ObjectWindows Programmer's Guide*.
- The Doc/View model and classes are discussed in Chapter 10 in the *ObjectWindows Programmer's Guide*.
- *TListBox* is discussed in Chapter 11 in the *ObjectWindows Programmer's Guide*.

Chapter

14

Making an OLE container

The next step in the ObjectWindows tutorial shows you how to make an OLE 2 container from the Drawing Pad application. Object Linking and Embedding (OLE) is an extension to Windows that lets the user seamlessly combine several applications into a single workspace. An OLE container application can host server objects, providing additional workspace where the user of your application can expand your application with the capabilities provided by OLE-server-enabled application.

The code for the example used in this chapter is contained in the files STEP14.CPP, STEP14DV.CPP, STEP14.RC, and STEP14DV.RC in the EXAMPLES/OWL/TUTORIAL directory where your compiler is installed.

How OLE works

Two different types of application are necessary for basic OLE operations:

- A *container* can have other applications or objects embedded within it, presenting the data from the embedded object as part of the container's own data set.
- A *server* can be embedded within a container application and can be used to manipulate the data that the server displays in the container's work space.

What is a container?

In this step of the tutorial, you'll make your Doc/View Drawing Pad application into an OLE container. Making Drawing Pad into an OLE container has some important ramifications: the application is no longer limited to displaying a set of lines, but can also display any kind of data that can be presented by any server users embed within their drawings. Although line drawing capability is still in the application and producing line drawings is still the main function of the application, users can now spice up their drawings with bitmaps, spreadsheet charts, even sound files.

Although you can do many of the same tasks by using the Clipboard to transfer data, it's easier to use OLE. Using the Clipboard, your application has to be able to accept the type of data stored there. This means if you want to accept bitmaps in the Drawing Pad application, you have to build the functionality required to accept and display bitmaps. This in no way prepares the application to accept spreadsheet charts, database tables, or or data in other graphic formats. To include another type of data requires implementing more functionality to interpret and display that data.

Using OLE, your application can display any type of data that is supported by an available OLE server. As far as your application is concerned, a bitmap looks exactly like a spreadsheet chart, a database table, or any other kind of object; that is, they all look like OLE server objects.

Also, using the Clipboard, you can build the ability to display a bitmap into your application. But modifying the bitmap after it's been pasted in requires more functionality to be built into your application.

Using OLE, the embedded server handles its embedded data whenever the user wants to modify or change it. The type of data used in the server is of no consequence to the container.

Implementing OLE in ObjectWindows: ObjectComponents

There is a price to pay for the advantages OLE provides for your application: programming an OLE implementation has historically been very messy and time consuming. You needed to modify your code to conform to OLE specifications. Even more than this, OLE doesn't follow the event-based paradigm that Windows applications were previoiusly based on. Instead it implements a new interface-based paradigm, requiring an understanding of standard OLE interfaces, reference counting, and other OLE specifications.

ObjectWindows implements OLE through the ObjectComponents Framework. You can use ObjectComponents to make your application an OLE container or server with only minor modifications to your code. You can use ObjectComponents with the following application types:

- Doc/View ObjectWindows applications
- Non-Doc/View ObjectWindows applications
- Non-ObjectWindows C++ applications

The fewest modifications are required for Doc/View ObjectWindows applications, which is shown in this chapter. Implementing OLE with ObjectComponents in non-Doc/View ObjectWindows applications and non-ObjectWindows C++ applications is described in Chapter 18 through Chapter 22 of the *ObjectWindows Programmer's Guide*.

The following steps are required to convert your Doc/View ObjectWindows application to an OLE container application:

- Include the proper header files.
- Register your application and Doc/View objects in the system registration database.
- Create a *TOcApp* object and associating it with your application object.
- Change your frame window class to an OLE-aware frame window class.

- Change your document and view classes's base class to OLE-enabled classes.

The ObjectComponents objects used in this chapter are explained as you add them to the Drawing Pad application. The ObjectComponents Framework is described in detail in Chapter 18 through Chapter 22 in the *ObjectWindows Programmer's Guide* and the *ObjectWindows Reference Guide*.

Adding OLE class header files

You need to add new headers to your files to use the ObjectComponents classes. ObjectComponents adds OLE capabilities by adding deriving new OLE-enabled classes from existing classes.

To add new headers to your files so you can use ObjectComponents classes:

1 In STEP14.CPP, instead of using *TDecoratedMDIFrame*, you'll use *TOleMDIFrame*, which is an OLE-enabled decorated MDI frame. All OLE frame windows, whether they're MDI or SDI, must be able to handle decorations, since many embedded OLE servers provide their own tool bars. The *TOleMDIFrame* class is declared in the owl/olemdifr.h header file.

   Your list of include statements in STEP14.CPP should now look something like this:

```
#include <owl/applicat.h>
#include <owl/dialog.h>
#include <owl/controlb.h>
#include <owl/buttonga.h>
#include <owl/statusba.h>
#include <owl/docmanag.h>
#include <owl/olemdifr.h>
#include <stdlib.h>
#include <string.h>
#include "step14.rc"
```

2 In STEP14DV.CPP, you need to include OLE-enabled document and view classes. These classes, *TOleDocument* and *TOleView*, provide standard Doc/View functionality along with the ability to support OLE. They're declared in the header files owl/oledoc.h and owl/oleview.h. Your list of include statements in STEP14DV.CPP should now look something like this:

```
#include <owl/chooseco.h>
#include <owl/dc.h>
#include <owl/docmanag.h>
#include <owl/gdiobjec.h>
#include <owl/inputdia.h>
#include <owl/listbox.h>
#include <owl/oledoc.h>
#include <owl/oleview.h>
#include <classlib/arrays.h>
#include "step14dv.rc"
```

Registering the application for OLE

For OLE to keep track of the applications running on a particular system, any application that wants to use OLE must register in the system-wide OLE registration database. You need to provide a unique identifier number and a description of the application. You also need to create objects that let your application communicate with OLE.

ObjectComponents simplifies the process of registering your application through a set of registration macros. These macros create an object of type *TRegList*, known as a registration table, which contains the information required by the OLE registration database. The macros are the same ones you use when creating a Doc/View template, but you use more of the capabilities available in the *TRegList* class. You can review how to create a table using these macros by seeing page 88.

Once you've created a registration table, you need to pass it to a connector object. A connector object provides the channel through which a Doc/View application communicates with ObjectComponents and, by extension, with OLE. The registration table is passed to an object of type *TOcApp* (ObjectComponents connector objects all begin with *TOc*).

Later, you'll modify the declaration of your *TDrawApp* class to be derived from both *TApplication* and *TOcModule*. Your application object initilizes the *TOcApp* connector object during the application object's construction. The connector object is then accessed through a pointer contained in the *TOcModule* class.

Creating the registration table

You use the REGDATA macro to create a container application's registration table. This is the same macro you used earlier to register your default document extension and file name filter. For your purposes now, you need the following key values:

Table 14.1 Key values and meanings

Key value	Meaning
clsid	String representation of a 16-byte number called a globally unique ID or GUID. This number must be unique to the application. It is used to distinguish your application from every other application on the system. This value is for internal system use only.
description	Application description for the system user to see. This string appears in the OLE registration list.

Your registration table should look something like this:

```
REGISTRATION_FORMAT_BUFFER(100)

BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid, "{383882A1-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"OWL Drawing Pad 2.0")
END_REGISTRATION
```

Note You must select a unique GUID for your application. There are a number of ways to get a unique identifier for your application. Generating a GUID and describing your

application is presented in detail in Chapter 20 of the *ObjectWindows Programmer's Guide*. For this tutorial, you can use the GUIDs provided in the tutorial examples. Do not use these same numbers when you create other applications.

Other macros can go into your registration table. Those for creating *AppReg* are the bare minimum for a container application object. You'll get to see a more complicated table when you create the registration table for your document class.

Also, because *AppReg* is created in the global name space of your application, it's safer and more informative to refer to it inside your classes and functions using the global scoping qualifier. So instead of:

```
void
MyClass::MyFunc()
{
  OtherFunc(AppReg);
}
```

you would write:

```
void
MyClass::MyFunc()
{
  OtherFunc(::AppReg);
}
```

Creating a class factory

A class factory is pretty much what it sounds like—it's an object that can make more objects. It is used in OLE to provide objects for linking and embedding. When an application wants to embed your application's objects in itself, it's the class factory that actually produces the embedded object.

ObjectWindows makes it easy to create a class factory with the *TOleDocViewFactory* template. All you need to do is create an instance of the template with the application class you want to produce as the template type. In this case, you want to produce instances of *TDrawApp* with your factory. Creating the template would look like this:

```
TOleDocViewFactory<TDrawApp>();
```

You need to pass an instance of this template as the second parameter of the *TOcRegistrar* constructor. You can see how this looks in the sample *OwlMain* below. The objects themselves are created in the factory using the same Doc/View templates used by your application when it's run as a stand-alone application.

TOleDocViewFactory is the class factory template for Doc/View ObjectWindows applications. There are other class factory templates for different types of applications. These are discussed in Chapter 19 of the *ObjectWindows Programmer's Guide* and in the *ObjectWindows Reference Guide*.

Creating a registrar object

The registration table contains information about your application object for the system. The registrar object, which is of type *TOcRegistrar*, takes the registration table and

registers the application with the OLE registration database. It also parses the application command line looking for OLE-related options.

To create a registrar object:

1 Create a global static pointer to a *TOcRegistrar* object. You can do this using the *TPointer* template, defined in the osl\geometry.h header file (this file is already included by a number of the ObjectWindows header files, so you don't need to include it again). This should look something like this:

```
static TPointer<TOcRegistrar> Registrar;
```

Using *TPointer* instead of a simple pointer, such as *TOcRegistrar* Registrar*, provides automatic deletion when the object referred to is destroyed or goes out of scope. The full range of operations available with regular pointers is available in *TPointer*, while some of the traditional dangers of using pointers are eliminated.

2 Create the actual registrar object. The *TOcRegistrar* constructor takes four parameters:

- A reference to a registration table object
- A pointer to a callback function of type *TComponentCreate*
- A string containing the application's command line
- An instance handle indicating the application instance the registrar is for; this parameter defaults to *_hInstance*, the current application instance

For these parameters, you can pass the following arguments when constructing the registrar object.

- For the first parameter, pass your registration table object.
- For the second parameter, pass in your class factory.
- For the third parameter, pass the application's command line. You can get the command line by calling *TApplication*'s *GetCmdLine* function.
- You don't need to specify the fourth parameter, an instance handle; just let that parameter take its default value.

For example,

```
::Registrar = new TOcRegistrar(AppReg, TOleDocViewFactory<TDrawApp>(),
                               TApplication::GetCmdLine());
```

3 Call the *Run* function. However, instead of calling the application object's *Run* function (which you couldn't do at this point if you wanted to, since you haven't created an application object), call the registrar object's *Run* function. *TOcRegistrar* provides a *Run* function that is called just like *TApplication*'s *Run* function. However, any ObjectWindows OLE application should call the registrar object's *Run* function. This function performs some checks and actions required for your OLE application.

Your *OwlMain* function should now look something like this:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  ::Registrar = new TOcRegistrar(AppReg, TOleDocViewFactory<TDrawApp>(),
                                 TApplication::GetCmdLine());
```

```
  return ::Registrar->Run();
}
```

Creating an application dictionary

The application dictionary is an object that helps coordinate associations between processes or tasks and *TApplication* pointers. Before diving into OLE, this was relatively simple: a *TApplication* object was pretty much synonymous with a process. With OLE, the environment becomes confused: there can be multiple tasks and processes in a single application, with a container application, a number of embedded servers, possibly more servers embedded within those servers—the neighborhood's gotten a little more crowded.

To deal with this, ObjectWindows provides application dictionaries with the *TAppDictionary* class. The best thing about *TAppDictionary* is that, in order to use it for our purposes here, you don't have to know a whole lot about it. ObjectWindows also provides a macro, DEFINE_APP_DICTIONARY, that creates and initializes an application dictionary object for you.

DEFINE_APP_DICTIONARY takes a single parameter, the name of the object you want to create. You should place this near the beginning of your source file in the global name space. You must at least place it before *TDrawApp*'s constructor, since that's where you'll use it.

Your application dictionary definition should look something like this:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Changes to TDrawApp

You need change the *TDrawApp* class to support ObjectComponents. These changes are fairly standard when you're creating a Doc/View application in an OLE container.

- Changing the class declaration
- Changing the class functionality, including
  - Creating an OLE MDI frame
  - Setting the OLE MDI frame's application connector
  - Adding a tool bar identifier

Changing the class declaration

You need to make the following changes to the declaration of the *TDrawApp* class:

1. Derive *TDrawApp* from both *TApplication* and the *TOcModule* class. *TOcModule* provides the interface your application object uses to communicate with OLE through the ObjectComponents Framework. Both *TApplication* and *TOcModule* should be public bases.
2. Change the constructor so that you pass the *TApplication* constructor a single parameter. You should initialize the name of the application object with the value of

description from *AppReg*. To make this easier, the *TRegList* class overloads the square bracket operators ([]) to return the string associated with the key value passed between the brackets. So to get the string associated with the *description* key, call `AppReg["description"]`.

Your *TDrawApp* declaration should now resemble the following code:

```
class TDrawApp : public TApplication, public TOcModule
{
  public:
    TDrawApp() : TApplication(::AppReg["description"]) {}

  protected:
    TMDIClient* Client;

    // Override methods of TApplication
    void InitInstance();
    void InitMainWindow();

    // Event handlers
    void EvNewView(TView& view);
    void EvCloseView(TView& view);
    void EvDropFiles(TDropInfo dropInfo);
    void CmAbout();

  DECLARE_RESPONSE_TABLE(TDrawApp);
};
```

Changing the class functionality

You need to change the main window to a *TOleMDIFrame* object and properly initialize it as follows:

- Creating an OLE MDI frame
- Setting the OLE MDI frame's application connector
- Adding a tool bar identifier

Creating an OLE MDI frame

Next, you need to change your *InitMainWindow* function by changing your frame window object from a decorated MDI frame to an *OLE-aware* decorated MDI frame (note that all OLE-aware ObjectWindows frame window classes are decorated). The window class to use for this is *TOleMDIFrame*. *TOleMDIFrame* is based on *TMDIFrame*, which provides MDI support, and *TOleFrame*, which provides the ability to work with ObjectComponents. Here's the constructor for *TOleMDIFrame*:

```
TOleMDIFrame(const char far* title,
             TResId menuResId,
             TMDIClient& clientWnd = *new TMDIClient,
             bool trackMenuSelection = false,
             TModule* module = 0);
```

The parameters to the *TOleMDIFrame* constructor are the same as those for *TDecoratedMDIFrame*. This makes the conversion simple: all you need to do is change the name of the class when you create the frame window object.

Setting the OLE MDI frame's application connector

In order for the OLE MDI frame to be able to handle embedded OLE objects, it needs to know how to communicate with the ObjectComponents mechanism. This is accessed through the *TOcApp* object associated with the application object. The frame window must be explicitly associated with this object.

To do this, *TOleMDIFrame* provides a function (inherited from *TOleFrame*) called *SetOcApp*. *SetOcApp* returns **void** and takes a pointer to a *TOcApp* object. For the parameter to *SetOcApp*, you can just pass *OcApp*.

Adding a tool bar identifier

OLE servers often provide their own tool bar to replace yours while the server is functioning. The mechanics of this are handled by ObjectComponents. But in order to put the server's tool bar in place of yours, ObjectWindows must be able to find your tool bar.

ObjectWindows tries to locate your tool bar by searching through the list of child windows owned by the OLE MDI frame window and checking each window's identifier. Up until now, your tool bar hasn't actually had an identifier, which would cause ObjectWindows to not find the tool bar. In order for ObjectWindows to identify the container's tool bar, the container must use the IDW_TOOLBAR as its window ID (the *Id* member of the tool bar's *Attr* member object).

Your *InitMainWindow* function should now look something like this:

```
void
TDrawApp::InitMainWindow()
{
  // Construct OLE-enabled MDI frame
  TOleMDIFrame* frame;
  frame = new TOleMDIFrame(GetName(), 0, *(Client = new TMDIClient), true);

  // Set the frame's OcApp to OcApp
  frame->SetOcApp(OcApp);

  // 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_FILENEW, CM_FILENEW, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS, TButtonGadget::Command));
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR, TButtonGadget::Command));
```

```
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT, TButtonGadget::Command));
  cb->SetHintMode(TGadgetWindow::EnterHints);

  // Set the control bar's id.  Required for OLE tool bar merging
  cb->Attr.Id = IDW_TOOLBAR;

  // 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);
  GetMainWindow()->SetMenuDescr(TMenuDescr("MDI_COMMANDS",1,1,0,0,1,1));

  // Install the document manager
  SetDocManager(new TDocManager(dmMDI | dmMenu));
}
```

Changes to the Doc/View classes

There are a number of changes you need to make to your *TDrawDocument* and *TDrawView* classes to support OLE containers. For your document class, you need to

- Add more information to the registration table for creating *TDrawDocument* document templates
- Change the base class to *TOleDocument*
- Modify the constructor
- Add two new functions, *GetLine* and *IsOpen*
- Modify the file access functions to store and load OLE objects

For your view class, you need to

- Change the base class to *TOleView*
- Remove the *DragDC* member
- Modify the constructor and destructor to remove statements with *DragDC*
- Modify the *Paint* function to call the base class *Paint* function
- Modify the mouse action commands to check when the user selects an embedded OLE object

These changes are described in the following sections.

Changing document registration

The registration table you created on page 88 contains information necessary for the creation of a basic document template. This functions fine when the only thing using the

document template is the document manager. But the way that ObjectComponents uses the Doc/View classes requires some more information:

- An identifier string. For this identifier, you want to use the REGDATA macro with the *progid* key value. This is a three part identifier. Each part of the identifier should be a text description, with each part separated by a period. There should be no whitespace or non-alphabetic character in this string other than the period delimiters.
  - The first part of the identifier should be descriptive of the overall application. For example, in the sample code, the first part of the identifier is DrawPad.
  - The second part should describe the part of the application contained in the module associated with the registration table. For the application registration table in the sample code, this part of the identifier is Application. For the document registration table, it's Document.
  - The third part should be a number. In the sample code, this number is 1. If your application supports multiple document types, use a different number for each document type.

  Note that this isn't meant for the users of your application to see. It's entered in the system's OLE registration database and should be unique for every application.
- A description of the document class. For this, you want to use the REGDATA macro with the *description* key value. This value is intended for the users of your application to see; this is the string that appears in the OLE registration database when someone is inserting an object into their container.
- A list of the types of data the container application can pass on to the Clipboard. To register Clipboard formats, use the REGFORMAT macro. This macro takes five parameters:
  - Format priority. The lower the value, the higher the priority. 0 indicates that the format is the highest priority format. When the user tries to paste data into your application, the Clipboard tries to paste it in as the highest priority format that is consistent with the format of the data in the Clipboard.
  - Data format.
  - Presentation aspect used to display data (for example, a bitmap could be displayed as a bitmap, as formatted information about the bitmap such as its dimensions and number of colors, as a hex dump, and so on) or an object might be presented in iconic form.
  - How the data is transferred when not otherwise specified (for example, when data is transferred by a drag-and-drop transaction, the server might prefer to pass the data to the container by means of a temporary file).
  - Whether the document can provide as well as receive this type of data.

  Every OLE application *must* specify that it can handle the *ocrEmbedSource* and *ocrMetafilePict* formats. By default, ObjectComponents always registers *ocrLinkSource*. You'll usually want to register *ocrLinkSource* yourself, though, so that you can set its priority lower. In addition, you can register *ocrBitmap* and *ocrDIB*. Note these formats indicate the type of data your application can pass to the Clipboard, not the type of data your application can accept. Pasting this data to the

Clipboard is handled by ObjectComponents. The exact meaning of each of these values is described in the *ObjectWindows Reference Guide*.

The following registration table shows how your registration table should look. The values for the REGFORMAT macro are described in the *ObjectWindows Reference Guide*.

```
BEGIN_REGISTRATION(DocReg)
  REGDATA(progid, "DrawContainer")
  REGDATA(description,"OWL Drawing Pad 2.0 Document")
  REGDATA(extension, "PTS")
  REGDATA(docfilter, "*.pts")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage,  ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict,    ocrGet)
  REGFORMAT(2, ocrBitmap, ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib, ocrContent,  ocrHGlobal|ocrStaticMed, ocrGet)
  REGFORMAT(4, ocrLinkSource, ocrContent,  ocrIStream,    ocrGet)
END_REGISTRATION
```

Changing TDrawDocument to handle embedded OLE objects

You need to make a few changes to *TDrawDocument* to support embedded OLE objects. These changes mainly affect reading and writing documents that contain OLE objects. The changes are fairly simple; most of the capabilities required to handle embedded OLE objects are handled in the new base class *TOleDocument*. Here's a summary of the changes required.

- Change *TDrawDocument*'s base class to *TOleDocument*.
- Modify *TDrawDocument*'s constructor to improve performance.
- Remove the *IsOpen* function.
- Add some function calls to the *Commit* and *Open* functions; these function calls read and write OLE objects embedded in the document.

Changing TDrawDocument's base class to TOleDocument

To get your document class ready to work in an ObjectComponents environment, you need to change the base class from *TFileDocument* to *TOleDocument*. *TOleDocument* is based on the *TStorageDocument* class, which is in turn based on *TDocument*. *TStorageDocument* provides the ability to manage and store compound documents. Compound documents provide a way to combine multiple objects into a single disk file, without having to worry about where each of the individual objects are stored or how they written out or read in. On top of *TStorageDocument*'s capabilities, *TOleDocument* adds the ability to interface with an OLE object, control and display the OLE object, and read and write the object to and from storage.

To change your base class from *TFileDocument* to *TOleDocument*, you first need to change all references from *TFileDocument* to *TOleDocument*. This is fairly simple, since all that needs to change is the actual name; all the function signatures, including the base class constructor's, are the same.

Constructing and destroying TDrawDocument

The only change you need to make to the constructor for *TDrawDocument* (other than changing the base class to *TOleDocument*) basically serves to enhance the performance of the Drawing Pad application, and is not connected to its OLE functionality.

1. Remove the *Lines* member from the constructor's initialization list.
2. Initialize *Lines* in the constructor body, with an initial size of 100, lower boundary of 0, and a delta of 5.

Your constructor should now look something like this:

```
TDrawDocument(TDocument* parent) : TOleDocument(parent), UndoLine(0), UndoState(UndoNone)
{
  Lines = new TLines(100, 0, 5);
}
```

You don't need to make any changes to the destructor.

Removing the IsOpen function

You need to remove the *IsOpen* function from the *TDrawDocument* class. This function is made obsolete by the change you made to the constructor, since the function tests the validity of the *Lines* member, and *Lines* now always points to a valid object.

TStorageDocument provides an *IsOpen* function that tests whether the document object has a valid *IStorage* member. *IStorage* is an OLE 2 construct that manages compound file storage and retrieval. A compound file is a basically a file that contains references to objects in a number of other locations. To the user, the compound file appears to be a single document. In reality, the different elements of the file are stored in various areas determined by the system and managed through the *IStorage* object. By constructing an OLE container, you're venturing into supporting compound documents in your application. However, since the support is provided through the OLE-enabled ObjectComponents classes, you don't need to worry about managing the compound documents yourself.

Along with removing the *IsOpen* function declaration and definition from *TDrawDocument*, you need to eliminate any references to the *IsOpen* function. This function is called only once, in the *GetLine* function. In this case, you can simply remove the entire statement that contains the call to *IsOpen*. This statement checks the validity of the document's *TLine* object referenced by the *Lines* data member, but the change you made to the constructor, which ensures that each document object is always associated with a valid *TLine* object, makes the check unnecessary. Your *GetLine* function should now look something like this:

```
TLine*
TDrawDocument::GetLine(uint index)
{
  return index < Lines->GetItemsInContainer() ? &(*Lines)[index] : 0;
}
```

The *TDrawDocument* class declaration should now look something like this:

```
class _DOCVIEWCLASS TDrawDocument : public TOleDocument
{
```

```
  public:
    enum {
      PrevProperty = TFileDocument::NextProperty-1,
      LineCount,
      Description,
      NextProperty,
    };
    enum {
      UndoNone,
      UndoDelete,
      UndoAppend,
      UndoModify
    };
    TDrawDocument(TDocument* parent = 0);
   ~TDrawDocument() { delete Lines; delete UndoLine; }

    // implement virtual methods of TDocument
    bool Open(int mode, const char far* path=0);
    bool Close();
    bool Commit(bool force = false);
    bool Revert(bool clear = false);

    int FindProperty(const char far* name);  // return index
    int PropertyFlags(int index);
    const char far* PropertyName(int index);
    int PropertyCount() {return NextProperty - 1;}
    int GetProperty(int index, void far* dest, int textlen=0);

    // data access functions
    TLine* GetLine(uint index);
    int AddLine(TLine& line);
    void DeleteLine(uint index);
    void ModifyLine(TLine& line, uint index);
    void Clear();
    void Undo();

  protected:
    TLines* Lines;
    TLine* UndoLine;
    int UndoState;
    int UndoIndex;
    string FileInfo;
};
```

Reading and writing embedded OLE objects

The last change you need to make to your document class provides the ability to save and load OLE objects embedded in a document. This is contained in two functions provided by *TOleDocument*. The functions are named *Open* and *Commit*. As you can probably guess, *Open* reads in the OLE objects contained in the document and *Commit* writes them out, that is, it commits the changes to disk.

To add these changes to your document class:

1 Add the call to *TOleDocument::Commit* in the *Commit* function right before you create the *TOutStream* object by calling the *OutStream* function.

2 Add the call to the *TOleDocument::Open* function in *TDrawDocument*'s *Open* function right before you create the *TInStream* object by calling the *InStream* function.

3 At the end of the procedure, call *TOleDocument::CommitTransactedStorage* to make your changes permanent. By default, *TOleDocument* uses the transacted mode (*ofTransacted*) to buffer changes in temporary storages until they are committed permanently.

That's all you need to do read and store OLE objects in your document! Your *Commit* function should now look something like this:

```
bool TDrawDocument::Commit(bool force)
{
  TOleDocument::Commit(force);

  TOutStream* os = OutStream(ofWrite);

  if (!os)
    return false;

  // Write the number of lines in the figure
  *os << Lines->GetItemsInContainer();

  // Append a description using a resource string
  *os << ' ' << FileInfo << '\n';

  // Get an iterator for the array of lines
  TLinesIterator i(*Lines);

  // While the iterator is valid (i.e. we haven't run out of lines)
  while (i)
    // Copy the current line from the iterator and increment the array.
    *os << i++;

  delete os;

  // Commit the storage if it was opened in transacted mode
  TOleDocument::CommitTransactedStorage();
  SetDirty(false);
  return true;
}
```

Your *Read* function should look something like this:

```
bool TDrawDocument::Open(int mode, const char far* path)
{
  char fileinfo[100];

  TOleDocument::Open(mode, path);
  if (GetDocPath()) {
    TInStream* is = (TInStream*)InStream(ofRead);
```

```
    if (!is)
      return false;

    unsigned numLines;
    *is >> numLines;
    is->getline(fileinfo, sizeof(fileinfo));

    while (numLines--) {
      TLine line;
      *is >> line;
      Lines->Add(line);
    }

    delete is;

     FileInfo = fileinfo;
  } else {
    FileInfo = string(*::Module,IDS_FILEINFO);
  }
  SetDirty(false);
  UndoState = UndoNone;
  return true;
}
```

Changing TDrawView to handle embedded OLE objects

You need to make a few changes to *TDrawView* to support embedded OLE objects. These changes mainly affect handling OLE objects through the mouse, including dragging the objects and activating the object's server. The changes are fairly simple; most of the capabilities required to handle embedded OLE objects are handled in the new base class *TOleView*. Here's a summary of the changes required.

- Change the base class of *TDrawView* to *TOleView*.
- Remove the *DragDC* member; *TOleView* supplies a *TDC* pointer called *DragDC*.
- Modify the constructor and destructor to remove initialization and deletion of *DragDC*.
- Remove the *EvRButtonDown* function.
- Modify the *Paint* function to call *TOleView::Paint* to force embedded objects to paint themselves.
- Modify the mouse action functions to deal with user interaction with embedded OLE objects.
- Modify the class declaration to reflect changes in the view class.

Modifying the TDrawView declaration

Here's the class declaration for *TDrawView*. The modifications to it will be explained in the following sections.

```
class _DOCVIEWCLASS TDrawView : public TOleView
{
  public:
    TDrawView(TDrawDocument& doc, TWindow* parent = 0);
   ~TDrawView() {delete Line;}
    static const char far* StaticName() {return "Draw View";}
    const char far* GetViewName() {return StaticName();}

  protected:
    TDrawDocument* DrawDoc;  // same as Doc member, but cast to derived class
    TPen* Pen;
    TLine* Line;     // To hold a single line sent or received from document

    // Message response functions
    void EvLButtonDown(uint, TPoint&);
    void EvMouseMove(uint, TPoint&);
    void EvLButtonUp(uint, TPoint&);
    void Paint(TDC&, bool, TRect&);
    void CmPenSize();
    void CmPenColor();
    void CmClear();
    void CmUndo();

    // Document notifications
    bool VnCommit(bool force);
    bool VnRevert(bool clear);
    bool VnAppend(uint index);
    bool VnDelete(uint index);
    bool VnModify(uint index);

  DECLARE_RESPONSE_TABLE(TDrawView);
};
```

Here's the response table for *TDrawView*.

```
DEFINE_RESPONSE_TABLE1(TDrawView, TOleView)
  EV_WM_LBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
  EV_COMMAND(CM_PENSIZE, CmPenSize),
  EV_COMMAND(CM_PENCOLOR, CmPenColor),
  EV_COMMAND(CM_EDITCLEAR, CmClear),
  EV_COMMAND(CM_EDITUNDO, CmUndo),
  EV_VN_COMMIT,
  EV_VN_REVERT,
  EV_VN_DRAWAPPEND,
  EV_VN_DRAWDELETE,
  EV_VN_DRAWMODIFY,
END_RESPONSE_TABLE;
```

Changing TDrawView's base class to TOleView

To get your view class ready to work in an ObjectComponents environment, you need to change the base class from *TWindowView* to *TOleView*. *TOleView* is itself based on the *TWindowView* class. *TOleView* provides the ability required to manipulate and move OLE objects and activate an object's server.

To change your base class from *TWindowView* to *TOleView*, you first need to change all references from *TWindowView* to *TOleView*. This is fairly simple, since all that needs to change is the actual name; all the function signatures, including the base class constructor's, are the same.

Removing DragDC

This change is relatively straightforward. *TOleView* provides a pointer to a *TDC* called *DragDC*, obviating the need for this member in the *TDrawView* class. You'll also need to remove a lot of the actions you previously took with *DragDC*. Many of these, such as creating a device context object when the left mouse button is clicked, is taken care by *TOleView*. These changes are discussed in the next section where they come up.

Constructing and destroying TDrawView

The only change you need to make to the *TDrawView* constructor is to remove the initialization of the *DragDC* member. Although *DragDC* was removed from the *TDrawView* class declaration, it is still a class member; it is provided by *TOleView*. But *TOleView* also handles initializing *DragDC*, since *TOleView* needs to check for OLE actions that the user might have taken.

Note that the *TOleView* constructor signature is the same as that of *TWindowView*, meaning all you have to do is change the name and nothing else. Here's how your *TDrawView* constructor should look.

```
TDrawView::TDrawView(TDrawDocument& doc, TWindow* parent) :
  TOleView(doc, parent), DrawDoc(&doc)
{
  Line = new TLine(TColor::Black, 1);
  SetViewMenu(new TMenuDescr(IDM_DRAWVIEW));
}
```

By the same token, the only modification needed to the destructor for *TDrawView* is to remove the statement deleting *DragDC*.

```
~TDrawView()
{
  delete Line;
}
```

Modifying the Paint function

You need to modify the *Paint* function to call *TOleView::Paint*. *TOleView::Paint* finds each linked or embedded object in the document (if there are any) and instructs each one to paint itself. Once this has been done, you can go on and paint the screen just as you did in Step 13. Your new *Paint* function should look something like this:

```
void
TDrawView::Paint(TDC& dc, bool erase, TRect&rect)
{
  TOleView::Paint(dc, erase, rect);

  // Iterates through the array of line objects.
  int j = 0;
  TLine* line;
  while ((line = const_cast<TLine *>(DrawDoc->GetLine(j++))) != 0)
    line->Draw(dc);
}
```

Selecting OLE objects

The next changes you need to make involve the functions dealing with mouse actions, namely *EvLButtonDown*, *EvMouseMove*, and *EvLButtonUp*. The changes you need to make in these functions involve checking whether the user's mouse actions involve an OLE object and what drawing mode is set. This is mostly handled by *TOleView*; for the most part, all you have to do is call the base class versions of the functions. The changes for each function are discussed in the following sections.

Modifying EvLButtonDown

You don't need to change the basic workings of the *EvLButtonDown* function as it exists in Step 13. What you do need to do is add a couple of extra steps to take into account OLE objects that might be in the view.

1. The first thing you need to do is let the *TOleView* base class determine whether the user selected an OLE object. Do this by calling *TOleView::EvLButtonDown*. This function deactivates any currently selected OLE object, creates a new *TOleDC* object (*TOleDC* is derived from the *TClientDC* class you used in previous steps, adding the ability to handle embedded OLE objects), and checks to see if another OLE object was selected.
2. To check whether the user wants to and is able to draw in the view, you need to check two things: whether a valid device context was created in the call to *TOleView::EvLButtonDown* and whether an OLE object was selected. You can check the validity of the device context simply by testing *DragDC*. You can find out whether an OLE object was selected by calling the *SelectEmbedded* function. *SelectEmbedded* returns **true** if an object was selected and **false** otherwise. If both these conditions weren't met, *EvLButtonDown* can just return.
3. Assuming there is a valid device context and no OLE object was selected, you can go ahead and begin the drawing operation the same as you did in Step 13. The only change you need to make is removing the initialization of *DragDC*, since it's already set to a valid device context object.

Your *EvLButtonDown* function should look something like this:

```
void TDrawView::EvLButtonDown(uint modKeys, TPoint& point)
{
  TOleView::EvLButtonDown(modKeys, point);

  if (DragDC && !SelectEmbedded()) {
```

```
    SetCapture();
    Pen = new TPen(Line->QueryColor(), Line->QueryPenSize());
    DragDC->SelectObject(*Pen);
    DragDC->MoveTo(point);
    Line->Add(point);
  }
}
```

Modifying EvMouseMove

The changes needed to *EvMouseMove* are similar to those required by *EvLButtonDown*.

1. Call the base class version of *EvMouseMove*.
2. Check whether the device context is valid and whether an OLE object was selected.
3. Continue the drawing operation the same way you did in Step 13.

Your *EvMouseMove* function should look something like this:

```
void TDrawView::EvMouseMove(uint modKeys, TPoint& point)
{
  TOleView::EvMouseMove(modKeys, point);

  if (DragDC && !SelectEmbedded()) {
    DragDC->LineTo(point);
    Line->Add(point);
  }
}
```

Modifying EvLButtonUp

With *EvLButtonUp*, you need to do the same things as you did in *EvLButtonDown* and *EvMouseMove*, but with a bit of a twist. In this case, call the base class version of the function last instead of first. *TOleView::EvLButtonUp* performs a number of cleanup operations, including deleting the device context object pointed to by *DragDC*.

1. Check whether the device context is valid and whether an OLE object was selected.
2. Perform the same operations as *EvLButtonUp* in Step 13, except for deleting and zeroing out *DragDC*.
3. Call *TOleView::EvLButtonUp*.

Your *EvLButtonUp* function should look something like this:

```
void TDrawView::EvLButtonUp(uint modKeys, TPoint& point)
{
  if (DragDC && !SelectEmbedded()) {
    ReleaseCapture();
    if (Line->GetItemsInContainer() > 1) {
      DrawDoc->AddLine(*Line);
    }
    Line->Flush();
    delete Pen;
  }
```

```
  TOleView::EvLButtonUp(modKeys, point);
}
```

Where to find more information

Here's a guide to where you can find more information on the topics introduced in this step:

- OLE and ObjectComponents containers are discussed in Chapter 19 in the *ObjectWindows Programmer's Guide.*
- The ObjectComponents classes in general are discussed in more detail in Chapters 18 through 22 in the *ObjectWindows Programmer's Guide.*

Chapter

15

Making an OLE server

Supporting OLE servers by being a OLE container is a big step ahead in flexibility for your applications. It expands the functionality of your application into just about any area you can think of. But one thing is missing: if you can make your application an OLE server, your application can be used to extend the functionality of other applications.

For example, suppose you're developing database forms and you want to add some of your line drawings to make the database forms more attractive. Without OLE, including line drawings in the database form is rather cumbersome, requiring you somehow to capture the drawing and paste it into the form. Then, once it's in the form, you have no way to modify it besides going back to Drawing Pad, editing it, then pasting it back into the form.

If the database is an OLE container, and you've made Drawing Pad an OLE server, you can easily drop line drawings into your database forms. The embedded OLE server lets you modify the line drawing without having to leave your database application.

This chapter describes how to take your Doc/View Drawing Pad application from Step 14 and make it an OLE server. The code for this example can be found in the files STEP15.CPP, STEP15DV.CPP, STEP15.H, STEP15DV.H, STEP15.RC, and STEP15DV.RC in the EXAMPLES/OWL/TUTORIAL directory of your compiler installation.

Note After making the changes in this step, the Drawing Pad application will be a server-only application; that is, it will no longer support containing embedded OLE objects. This is to demonstrate the unique server functionality added to the application. Changes that remove the container support will be noted. If you want to combine container and server support in a single application, you need only to skip those steps that remove container support.

Converting your application object

There are a few changes you need to make in your application object to become an OLE server.

- Change the header files.
- Change the application's registration table.
- Change the base class constructor to register some more information, including the application dictionary.
- Hide the window if the application was invoked as a server.
- Add module identifier parameters to a number of object constructors.
- Change how you create new views.
- Change how you find the About dialog box's parent window.
- Change the *OwlMain* function to check for action options.

Changing the header files

You only need to make two changes to the list of header files in STEP15.CPP.

- Add the owl\oleview.h header file; the *TOleView* class needs to be used when you create new views
- Change from including STEP14.RC to STEP15.RC

Changing the application's registration table

You basically need to change your entire application registration table from Step 14. However, only one of these changes is directly related to making the application an OLE server. You need to change the values associated with the *clsid* and *description* keys. Because the end result is an application that is different from Step 14, all of these values should change.

Your new registration table should look something like this:

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid, "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"OWL Drawing Pad Server")
END_REGISTRATION
```

Note Remember, don't try to duplicate the GUID or program identifier in your other applications! Preventing such duplication is why these values were changed from Step 14 to Step 15!

Changing the application constructor

For OLE servers, you need to change *TDrawApp*'s base class constructor to take the application dictionary object as a parameter. For a container application, you didn't

need to do this. The reason is that a container is always be created as an executable application as opposed to a DLL. When you don't specify an application dictionary in *TApplication*'s constructor, it uses the global application dictionary *::OwlAppDictionary*. This works fine for an executable: since it has its own instance, it's entered in the global application dictionary. But DLLs don't have their own instance.

TApplication provides a couple more parameters to its constructor than you've been using. The first is the name of the application, which you used in the last step to set the application name.

The second is a pointer to a reference to a *TModule* object (that is, *TModule*&*). *TApplication*'s constructor sets this pointer to point at the new application object. In this case, you want to pass in the global module object *::Module*. *::Module* is used by ObjectWindows and ObjectComponents to identify the current module. Note that *::Module* is the default value for this parameter.

The last parameter is a pointer to a *TAppDictionary* object. Use a pointer to the *TAppDictionary* object you created using the DEFINE_APP_DICTIONARY macro for this parameter.

Now your constructor should look something like this:

```
TDrawApp() : TApplication(::AppReg["description"], ::Module, &::AppDictionary) {}
```

Hiding a server's main window

Under regular circumstances, when your application is started up, it does some setup and initialization, then creates a main window for the user to work in. That's fine when someone is using your application as their primary workplace. But when your application is being used as an OLE server, it's *not* the primary workplace; the main window has already been created by another application. In this case, you need to set your main window to be hidden.

The best place to do this is in *InitMainWindow*, before your window object has been created. To find out whether the application is an embedded server and to hide the main window if so:

1. Call the *IsOptionSet* function of the *TOcRegistrar* object, passing *TOcCmdLine::Embedding* as the function's argument. You can get a reference to the application's registrar object by calling the *GetRegistrar* function. *IsOptionSet* checks to see if the application's command line contained the option passed to it as a parameter. When an application is created as an embedded server, the *-Embedding* option is specified on the command line. Therefore, if the application was created as an embedded server, *IsOptionSet* returns **true** when passed *TOcCmLine::Embedding*. You'll see more of these options later.
2. If *IsOptionSet* returns **true**, the application is being invoked as an embedded server, so set *nCmdShow* to SW_HIDE. This causes the main window to be hidden when it's created and activation to be passed to the window from which the server was invoked.

Identifying the module

When you constructed the *TApplication* base class, you had to add in a couple of new parameters to make sure the object could find itself in complicated OLE environment. You need to do the same basic thing for a number of other objects in your application. In the case of these objects, though, you just need to direct them to the application object, which then handles all the transactions between your application and whatever's outside of the application.

- The MDI client window takes a single parameter, a module pointer.
- The OLE MDI frame takes a *TModule* pointer as a parameter after its menu-tracking parameter (which is the last parameter you used in Step 14).
- The document manager takes a *TApplication* pointer after its flags parameter.

Use *TDrawApp*'s **this** pointer for each of these parameters.

Your *InitMainWindow* function should look something like this:

```
void
TDrawApp::InitMainWindow()
{
  if (GetRegistrar().IsOptionSet(TOcCmdLine::Embedding))
    nCmdShow = SW_HIDE;

  TOleMDIFrame* frame;
  frame = new TOleMDIFrame(GetName(), 0, *(Client = new TMDIClient(this)), true, this);
  frame->SetOcApp(OcApp);

  // 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_FILENEW, CM_FILENEW, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS, TButtonGadget::Command));
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE, TButtonGadget::Command));
  cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR, TButtonGadget::Command));
  cb->Insert(*new TSeparatorGadget);
  cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT, TButtonGadget::Command));
  cb->SetHintMode(TGadgetWindow::EnterHints);
  cb->Attr.Id = IDW_TOOLBAR;

  // 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);
  GetMainWindow()->SetMenuDescr(TMenuDescr(IDM_MDICMNDS));
```

```
    // Install the document manager
    SetDocManager(new TDocManager(dmMDI | dmMenu, this));
}
```

Creating new views

When creating a new view window in an OLE server application, you need to be careful about setting the view's parent. In the case where your application is being run as a stand-alone program, you don't have to change anything. The code in *EvNewView* that you used in the last few steps is just fine.

Things become complicated when the server is embedded in a container application. You need to determine one basic thing: is your view using space inside one of the container's windows? You can determine this by answering two questions:

- Is the application being used as an embedded server? If the answer to this question is no (that is, your application is being run on its own), then you can skip the next question: you know your application isn't occupying space in the container's window, because there is no container.
- Has the application been opened for editing? The user can access your embedded server in one of two ways: either in-place editing, where your server's workspace sits inside the workspace of the container, or open editing, where your server opens up for editing, looking pretty much the same as it does when opened on its own. If the user has opened your server for editing then the server is *not* sharing space in the container's window. Only if the user is using your server for in-place editing do you have to worry about sharing space with the container.

The reason you need to determine this has to do with setting the parent window of the view. When the server is being used as an in-place server, you must set the parent window of the view properly. ObjectComponents provides an object known as a view bucket to make this easier. Once you've set your view's parent to the view bucket, ObjectComponents takes care of setting the view's parent when the view is activated, deactivated, moved around, and so on. To set the view's parent, follow this procedure:

1. Downcast the *TView* parameter of the *EvNewView* function to a *TOleView*. Take the address of the object by prefixing it with an ampersand (&) and assign it to a *TOleView* pointer using the TYPESAFE_DOWNCAST macro.
2. Check whether the view is an embedded server by calling the view's associated document's *IsEmbedded* function. The view itself doesn't know if it's embedded. You can find the view's associated document by calling the view's *GetDocument* function. If the document's not embedded, you can stop checking here and just go to the code you used in the last few steps.
3. Check whether the view is activated for open editing. You can check this by calling the *IsOpenEditing* function of the view's remote view. You can get a pointer to the remote view by calling *GetOcRemView*. If *IsOpenEditing* returns **true**, you can stop checking here and go to the code you used in the last few steps.
4. Once you've determined that the application is being used as a server for in-place editing, you can work on setting up the view's parent. Follow this procedure:

1. Find the window associated with the view. You can get a *TWindow* pointer to this window using the view's *GetWindow* function.
2. You need to find the remote view bucket associated with the server. To do this, call the *GetMainWindow* function and downcast the return value to a *TOleFrame* pointer. *TOleFrame* provides a function called *GetRemViewBucket*. This function returns a *TWindow* pointer that references the remote view bucket.
3. Once you've found the remote view bucket, call the view's *SetParent* function with the bucket's *TWindow* pointer as the parameter.
4. Call the view's *Create* function.

Note that you haven't really set the view's parent as you normally think of it. But the remote view bucket lets you set this once and then lets ObjectComponents take care of the work of keeping track of the active parent window.

The code for this function should look something like this:

```
void
TDrawApp::EvNewView(TView& view)
{
  TOleView* ov = TYPESAFE_DOWNCAST(&view, TOleView);
  if (view.GetDocument().IsEmbedded() && !ov->GetOcRemView()->IsOpenEditing()) {
    TWindow* vw = view.GetWindow();
    vw->SetParent(TYPESAFE_DOWNCAST(GetMainWindow(), TOleFrame)->GetRemViewBucket());
    vw->Create();
  } else {
    TMDIChild* child = new TMDIChild(*Client, 0);
    if (view.GetViewMenu())
      child->SetMenuDescr(*view.GetViewMenu());
    child->Create();
    child->SetClientWindow(view.GetWindow());
  }
}
```

Changing the About dialog box's parent window

In previous versions of the tutorial application, when you created the About dialog box, you simply called the *GetMainWindow* function to find the dialog box's parent window. This is no longer adequate, however, since you don't know if your main window is actually the main window that the application user sees. If your application is embedded in another application, you've already determined in the *TDrawApp* constructor that you're not displaying your main window.

To find the window with focus or other appropriate view window on the desktop (which functions as the dialog's parent), you can call the *GetCommandTarget* function. This function is provided by *TFrameWindow* and returns a handle to the current active window. Note that calling this function works whether or not the application is running as an embedded server or as a stand-alone application, since it returns the command focus window. When the tutorial application is an embedded server, it returns a handle to the focus window of the client application. When the tutorial application is running on its own, it returns a handle to itself.

Note that you still need to call *GetMainWindow* to get a pointer to the tutorial application's main window. You then call the *GetCommandTarget* function of that window object. You also need to create a temporary *TWindow* to pass *GetCommandTarget*'s return value to the *TDialog* constructor. Your modified *CmAbout* function should look something like this:

```
void
TDrawApp::CmAbout()
{
  TDialog(&TWindow(GetMainWindow()->GetCommandTarget()), IDD_ABOUT).Execute();
}
```

Modifying OwlMain

There's only one new thing you need to take care of before running an OLE server application. You need to check the command line to see if one of the standard action options was specified.

There are a couple of standard ObjectComponents command-line options that may be specified for your server application. The presence of one of these "action" options signals that, instead of executing normally, your application should perform a particular action, then exit. For an OLE server application, the action options you need to check for are:

- The *–RegServer* option tells your application to completely register itself in the OLE registration database.
- The *–UnregServer* option tells your application to "unregister" itself, that is, remove its entry in the OLE registration database.

The good thing about these options is that ObjectComponents automatically performs these actions for you when you create the registrar object. The only thing you need to do is check in the *OwlMain* function whether one of these options was set. If so, you can return immediately. If none of the action options was specified, you can go on to the next step.

You can check for these options using the *IsOptionSet* function that you used in the *InitMainWindow* function to check for the *–Embedding* flag. For these options, you should check for the *TOcCmdLine::AnyRegOptions* flag. This flag checks to see if any of the options relevant to your application was set. *IsOptionSet* returns **true** if any of the options was set.

If one of the flags was set, you can return 0 from *OwlMain*. When one of these action options is set, ObjectComponents performs some registration task. Once that task is done, the application is complete. Your application never performs a registration task then executes as normal.

Your *OwlMain* function should look something like this:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  Registrar = new TOcRegistrar(AppReg, TOleFactory<TDrawApp>(),
```

```
                                    TApplication::GetCmdLine());

  if (Registrar->IsOptionSet(TOcCmdLine::RegServer | TOcCmdLine::UnregServer))
    return 0;

  return Registrar->Run();
}
```

Changes to your Doc/View classes

There are a number of changes you need to make to your Doc/View classes to support OLE server functionality:

- Change your header files
- Modify the document registration table to provide extra information needed by an OLE server
- Make some changes to the view notification functions *VnRevert*, *VnAppend*, *VnModify*, and *VnDelete* functions
- Add some new members to *TDrawView*, including a *TControlBar* pointer and some new functions
- Remove calls from the mouse action functions and the *Paint* function

Changing header files

You need to change your list of header files to include a few new header files, along with changing to including the resource script file for Step 15. The new files you need to include are owl/controlb.h and owl/buttonga.h. Your include statements should look something like this:

```
#include <owl/dc.h>
#include <owl/inputdia.h>
#include <owl/chooseco.h>
#include <owl/gdiobjec.h>
#include <owl/docmanag.h>
#include <owl/listbox.h>
#include <owl/controlb.h>
#include <owl/buttonga.h>
#include <owl/olemdifr.h>
#include <owl/oledoc.h>
#include <owl/oleview.h>
#include <classlib/arrays.h>
#include "step15dv.rc"
```

Changing the document registration table

You need to make some fairly extensive changes to your document registration table to support being an OLE server. The parts that don't change are discussed in this section.

Defining the registration table isn't different from before. This basically involves using the BEGIN_REGISTRATION and END_REGISTRATION macros. As before, your table begins with the BEGIN_REGISTRATION macro, which takes the name of the registration as its only parameter. The END_REGISTRATION macro closes out the table definition.

The two REGDATA macros that set the *extension* and *docfilter* table entries remain the same. The REGDOCFLAGS macro also doesn't change.

The parts of the registration table that you need to change are discussed in the next sections.

Program identifier and description

Step 14's program identifier (the value associated with the *progid* key) and its description (the value associated with the *description* key) described the application as a "Draw Container" and "OWL Drawing Pad Container" respectively. These values need to be changed to reflect the application being a server.

Making the application insertable

ObjectComponents provides a special key value called *insertable*. You can register *insertable* using the REGDATA macro. The value associated with the *insertable* key is irrelevant; it's never used, so usually you'll just want to set an empty string for the value.

The presence of the *insertable* key indicates to ObjectComponents that the application is insertable, that is, the application can be embedded into other applications. All ObjectComponents servers must specify the *insertable* key in their registration table!

Setting the server's menu items

When the user activates a server embedded in a container by clicking on the server's view, the container sets a menu item (usually on its Edit menu) that the user can use to access the server. This menu goes to a pop-up menu that provides a number of "verbs"—menu choices that let the user work with the server application and manipulate the data in it.

So there are two things you need to set up for this:

- You need to set up the menu name that the container uses to represent your application on the container's Edit menu. You can do this with the REGDATA macro, using the *menuname* key and the text you want to appear on the menu as the key's value. You want to be considerate of the container application when choosing this name. Use a name that you would normally use in a menu; that is, it should be descriptive of your application but not so long that it forces the menu to be quite large to accommodate the string. In this case, you could use the application name "Drawing Pad."
- You can specify up to twenty verbs for your server application. Specify the verbs for your server application using the REGDATA macro. The key values you use to set up verbs follow the format *verb*n, where n is a number from 0 to 19. The value you associate with each verb is the text that appears on the pop-up menu. Note that you can specify a keyboard shortcut for each verb by preceding the shortcut letter with an

ampersand (&). For example, if you specify Edit as a verb, and you want the user to be able to press E to activate that, you specify the string "&Edit" for the value.

Note that the first verb in the verb list, that is, the value associated with the *verb0* key is the default verb for your server. Thus if the user double-clicks on your embedded server, the server acts just the same as if the user had selected the *verb0* value from the server's menu.

ObjectComponents servers are set up to automatically handle two verbs.

- The Edit verb indicates that the user wants to manipulate the data handled by the server in place in the container. That means that the user works with the data right in the remote view area in the container's window.
- The Open verb indicates that the user wants to open the server application to manipulate the data. In this case, the application opens up as if the user had run the application by itself. The main difference between using the server this way and running the server as a stand-alone application is that the server writes to a document file provided by the container; the container's compound document storage handles the details of saving the data to disk.

Specifying Clipboard formats

For the server application, you can trim down the number of Clipboard formats available. You really only need to provide two formats.

- *ocrEmbedSource* indicates that the server can be copied to the Clipboard as an embeddable source. If someone tries to paste an embeddable source from the Clipboard, they get a copy of the embedded server object in their application.
- *ocrMetafilePict* indicates that the server can be copied to the Clipboard as a metafile representation.

As before, the actual copying operation is handled by ObjectComponents. Note that these are the only formats necessary to support an ObjectComponents server; the other formats provided by the container application are removed. To support dual container/server functionality, you should leave these formats in.

Your finished document registration table should look something like this:

```
BEGIN_REGISTRATION(DocReg)
  REGDATA(progid, "DrawServer")
  REGDATA(menuname, "Drawing Pad")
  REGDATA(description, "OWL Drawing Pad Server")
  REGDATA(extension, "PTS")
  REGDATA(docfilter, "*.pts")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt
  REGDATA(insertable, "")
  REGDATA(verb0, "&Edit")
  REGDATA(verb1, "&Open")
  REGFORMAT(0, ocrEmbedSource, ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict, ocrGet)
END_REGISTRATION
```

Changing the view notification functions

You need to make a change to a number of the view notification functions to support proper painting of the server's remote view. The functions you need to change are *VnRevert*, *VnAppend*, *VnModify*, and *VnDelete*. Each of these view notifications indicates that the drawing has been modified in some way and the display needs to be updated.

To force the container to update the view and reflect the changes in the view's appearance, you need to call the *InvalidatePart* function. This function is provided by *TDrawView*'s base class *TOleView*. This function tells the container window that the area inside the embedded server's remote view is invalid and needs repainting. *InvalidatePart* takes a single parameter, a *TOcInvalidate* **enum**. A *TOcValidate* can be one of two values.

- *invData* indicates the data in an embedded object has changed and should be updated in the container.
- *invView* indicates the appearance of an object has changed and should be updated in the container.

In this case, each of these view notification events indicates that the appearance of the drawing has changed, whether it was by discarding changes, appending a new line, modifying one of the current lines, or deleting a line. So when you do call the *InvalidatePart* function, you should call it with the *invView* argument. The *invData* argument is used when the container has a link to data in the server, but the container actually takes care of displaying the data.

You should first call the *Invalidate* function of the view when applicable (each of these functions already calls *Invalidate*, except for *VnAppend*, which doesn't need to), then call the *InvalidatePart* function. Here's how your modified view notification functions should look:

```
bool
TDrawView::VnRevert(bool /*clear*/)
{
  Invalidate();  // force full repaint
  InvalidatePart(invView);
  return true;
}

bool
TDrawView::VnAppend(uint)
{
  InvalidatePart(invView);
  return true;
}

bool
TDrawView::VnModify(uint /*index*/)
{
  Invalidate();  // force full repaint
  InvalidatePart(invView);
  return true;
}
```

```
bool
TDrawView::VnDelete(uint /*index*/)
{
  Invalidate();  // force full repaint
  InvalidatePart(invView);
  return true;
}
```

Adding new members to TDrawView

You need to add some new members to your *TDrawView* class. These members are

- A *TControlBar* pointer
- Two new event handlers for cutting and copying
- Two new event handlers for ObjectComponents events

Adding a control bar

When your application is activated as an embedded server, the container often lets the application provide a tool bar to access its functionality. This tool bar should be different from the regular application tool bar and provides button gadgets only to access the unique functions of your application and not those things handled by containers, that is, the object's editing and viewing commands. For example, opening a file is handled by any adequate container application, so it's not a unique ability of the Drawing Pad application. On the other hand, no container knows how to change Drawing Pad's pen color.

Since the commands supported by this tool bar are a subset of the commands supported by the application's tool bar, you can't simply use that tool bar. Instead you need to provide one for each embedded server view. To support this, just add a *TControlBar* pointer as a protected data member. You should initialize this member to 0 in *TDrawView*'s constructor. The tool bar itself is constructed in one of the new ObjectComponents event handlers. You can see this on page 150.

Cutting and copying data

Your server will often receive requests to cut or copy data to the Clipboard. You need to provide functions to handle these requests.

Cutting

Cutting data is copying information from the drawing, placing that information in the Clipboard, then removing the information from the drawing. This is a fairly common way to exchange data. However, in the context of the Drawing Pad application, this behavior is undefined: what does it mean to cut lines from a window?

But since this is a very common (almost mandatory) function in an OLE server, you should provide at least a place holder for it. You can declare and define a function called *CmEditCut* to do this. This function is called when *TDrawView* receives the CM_EDITCUT event, which you also need to add (it's in the STEP15DV.RC file in the sample code). So follow this procedure:

1 Add the CM_EDITCUT macro to your application.

2 Add the *CmEditCut* function to the *TDrawView* class declaration.

3 Add an EV_COMMAND macro to the response table to call *CmEditCut* when the CM_EDITCUT event is received.

4 Define *CmEditCut* to have no functionality.

Copying

To copy, you can call a member function of one of the classes provided by ObjectComponents. This class is called *TOcRemView* and provides a remote view object for a server document. A remote view handles the view of your server application from the container application. *TOcRemView* provides a function called *Copy*, which copies the document's data to the Clipboard. You get the *TOcRemView* object to work with by calling the *GetOcRemView* function, which is provided by *TOleView*.

1 Add the CM_EDITCOPY macro to your application.

2 Add the *CmEditCopy* function to the *TDrawView* class declaration.

3 Add an EV_COMMAND macro to the response table to call *CmEditCopy* when the CM_EDITCOPY event is received.

4 Define *CmEditCopy* to call *GetOcRemView* and call the *Copy* function of the *TOcRemView* object.

Handling ObjectComponents events

There are a couple of ObjectComponents events that you need to handle.

- OC_VIEWPARTSIZE indicates a request from the container to find out the size of your object's view, that is, the size of the "window" within the container's window in which the user sees your embedded application.
- OC_VIEWSHOWTOOLS indicates a request from the container for a tool bar from the server application.

Reporting server view size

For formatting reasons, a container often needs to find out the size of an embedded server's view. The container signals that it needs this information by sending an ObjectComponents message to the view. The view then needs to calculate the size of the server view and get that information back to the container.

To add this functionality, follow these steps:

1 The container lets the server know that it needs the size of the view by sending the OC_VIEWPARTSIZE, a standard ObjectComponents event. ObjectWindows provides a response table macro for this and other standard ObjectComponents event. The ObjectComponents event macros are defined in the header file owl/ocfevent.h, which is automatically included. These macros add EV_ to the beginning of the ObjectComponents event name, so that in this case the macro would be EV_OC_VIEWPARTSIZE. Add this macro to your view's response table. Like other

standard message macros, it has no parameters and calls a predefined function name when the event is received.

2 Add a function to your *TDrawView* class declaration to handle this event. The function called through the predefined response table macro is *EvOcViewPartSize*. This function returns **bool** and takes a pointer to a *TRect*.

3 Define the *EvOcViewPartSize* function. To do this, create a device context object (in the sample code here, we've used a *TClientDC*). You should place the size of the view in the *TRect* object passed into *EvOcViewPartSize* by pointer. In the Drawing Pad application, the size of the view is limited to 2 inches on the screen. This is an arbitrary measurement; you can also calculate the area necessary to display the information in the document and pass that back. For simplicity, though, it's easiest to pass back an absolute measurement. In this case, set the *top* and *left* members of the *TRect* to 0. You can then get the number of pixels in the size of the view by calling the *GetDeviceCaps* function of the device context object with the LOGPIXELSX parameter to get the width and the LOGPIXELSY parameter to get the height. This actually returns the number of pixels in an inch on the screen. Multiply this result by two in each case and assign the width to the *right* member of the *TRect* object and the height to the *bottom* member.

The completed function should look something like this:

```
bool
TDrawView::EvOcViewPartSize(TRect far* size)
{
  TClientDC dc(*this);

  // a 2" x 2" extent for server
  size->top    = size->left = 0;
  size->right  = dc.GetDeviceCaps(LOGPIXELSX) * 2;
  size->bottom = dc.GetDeviceCaps(LOGPIXELSY) * 2;
  return true;
}
```

Setting up the view's tool bar

The OC_VIEWSHOWTOOLS event indicates that the container in which your server is embedded wants to either show or hide your server's tool bar.

1 Add the OC_VIEWSHOWTOOLS macro to your view's response table.

2 Add a function to your *TDrawView* class declaration to handle this event. The function called through the predefined response table macro is *EvOcViewShowTools*. This function returns **bool** and takes reference to a *TOcToolBarInfo* object. *TOcToolBarInfo* is a simple structure; it only has a couple of members that we're concerned with here.

- The first is the *Show* member, a **bool**. If *Show* is **true**, the container wants to display your tool bar. If *Show* is **false**, the container wants to hide your tool bar.
- The second is *HTopTB*, an HWND. You pass back the tool bar to the container through this member.

3 If the container wants to hide the tool bar (that is, *Show* is **false**), you need to destroy the tool bar window, delete the tool bar object, and set your *TControlBar* pointer to 0. Before doing this, though, you should check to make sure that the *TControlBar* pointer references a valid object!

4 If the container wants to show the tool bar, you should first check to see that the *TControlBar* pointer doesn't already point to a valid object. If so, you can skip the next step and go on to step 6.

5 The most complicated thing about constructing a tool bar in these circumstances is finding the parent window. This takes a few steps, since you need to find your main window, and then, through the main window, which is an OLE frame window, you need to find the remote view bucket the application is using in the container's window.

   1 The first step is to find the application object. This is the easiest way to find the main window, since the application object provides a function to get a pointer to the main window. To find the application object, call the *GetApplication* function. This returns a *TApplication* pointer to the application object.

   2 Once you've found the application object, you can get a *TFrameWindow* pointer to the main window by calling the *GetMainWindow* function of the application object.

   3 Now that you've found the main window, you need to cast it to a *TOleFrame* window to be able to find the remote view bucket window. Although the main window is already a *TOleFrame* object, *GetMainWindow* returns it as a *TFrameWindow*. Since you are downcasting (that is, casting from a base object to a class derived from that base), you need to be careful. It is quite possible to try to cast an object of one type to an object of another type. If both of these types are derived from the same base class, this can cause serious trouble.

      For example, suppose you have a function that takes a *TWindow* pointer as its only parameter. When the function is called, you assume that the TWindow value you received in the function actually referenced a *TControl* object (since *TControl* is derived from *TWindow*, you can safely pass a *TControl* object as a *TWindow* object). But *TControl* and *TFrameWindow* are both derived from *TWindow*. What if the object passed in was actually a *TFrameWindow* object? Serious havoc could ensue.

      ObjectWindows provides a macro called TYPESAFE_DOWNCAST that downcasts objects that are typed as a base class to objects of a derived type. If the downcast isn't typesafe (that is, the object isn't what you're actually trying to downcast to, such as trying to cast a *TFrameWindow* to a *TControl*), the macro returns 0. Otherwise the macro makes the cast for you and returns the appropriate value.

      TYPESAFE_DOWNCAST takes two parameters. The first is the object you want to cast and the second is the type you want to cast the object to.

   4 Once you've found the application's main window and cast it appropriately, you need to call the *GetRemViewBucket* function. This function returns a *TWindow* pointer that references the remote view bucket window. This is quite important: with the tool bar parented properly, it's easy for the container to switch tool bars automatically among any of the servers that might be embedded in the container.

5 Once you've got a pointer to the remote view bucket window, construct a *TControlBar* object like normal, passing the view pointer as the parent. When the tool bar object is constructed, you can insert button gadgets to control the application. For now, it's sufficient to just add the CM_PENSIZE and CM_PENCOLOR buttons.

6 Once you have a valid tool bar object, create the tool bar itself by calling the object's *Create* function.

7 Once the tool bar is created, cast it to an HWND and assign it to the *TOcToolBarInfo*'s *HTopTB* member. You could instead assign it to one of *TOcToolBarInfo*'s other members to place it somewhere besides the top of the container's window.

8 Assuming everything went alright during this process, return **true**. This lets the container know that everything went alright and it can display the tool bar.

Here's how your *EvOcViewShowTools* function should look:

```
bool
TDrawView::EvOcViewShowTools(TOcToolBarInfo far& tbi)
{
  // Construct & create a control bar for show, destroy our bar for hide
  if (tbi.Show) {
    if (!ToolBar) {
      TOleFrame* frame = TYPESAFE_DOWNCAST(GetApplication()->GetMainWindow(), TOleFrame);
      ToolBar = new TControlBar(frame->GetRemViewBucket());
      ToolBar->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE, TButtonGadget::Command));
      ToolBar->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR,
                      TButtonGadget::Command));
    }
    ToolBar->Create();
    tbi.HTopTB = (HWND)*ToolBar;
  } else {
    if (ToolBar) {
      ToolBar->Destroy();
      delete ToolBar;
      ToolBar = 0;
    }
  }
  return true;
}
```

Removing calls from the Paint and mouse action functions

TDrawView's *Paint* function and its mouse action functions *EvLButtonDown*, *EvLButtonUp*, and *EvMouseMove* all make calls that are necessary to support container functionality. You should remove these calls for your application to function as a server-only application.

- The *Paint* function calls *TOleView::Paint* so that any embedded objects are called and told to paint themselves. Since a server-only application has no embedded objects, this call is no longer necessary.

- *EvLButtonDown*, *EvLButtonUp*, and *EvMouseMove* call the *SelectEmbedded* function to determine whether the user clicked on—and thereby selected—an embedded object. As with Paint, since there are no embedded objects in a server-only application, this call is no longer necessary.

Chapter

16

For further study

As you can see, ObjectWindows 2.5 packs a lot of functionality into its classes. With this tutorial, you've really only begun to scratch the surface of the things you can do with ObjectWindows. Here are a number of suggestions for things you can do to expand the tutorial application even more:

- You can add other Doc/View classes to the application. To do this, compile the document class, its view classes, and a list of document templates into an object file. Then add that object file to the application when you link it. Then, when you open a new document, you'll see the new document types appear in the File Open dialog box. Note that this works even though the application knows nothing about the Doc/View classes you added.
- A good source for Doc/View classes is the DOCVIEWX application in the EXAMPLES\OWL\OWLAPI\DOCVIEW directory. You can also try writing your own document and view classes.
- Try adding new GDI objects to the application. For example, you might try adding the ability to import bitmaps with the *TBitmap* class. Or add textured brushes with the *TBrush* class.
- Add different drawing operations, such as lines, boxes, circles, and so on. You can add menu choices for each of these operations. You can also set up exclusive state button gadgets on the control bar to let the user change the current operation just by pressing a button gadget.
- Try converting the control bar into a floating tool box by changing the *TControlBar* into a *TToolBox* in a *TFloatingFrame*. You can see an example of how this is done in the PAINT example in the EXAMPLES\OWL\OWLAPPS\PAINT directory.
- Try adding the ability to perform multiple undo operations. You can use container classes to hold all the lines that have been changed.

There are some additional steps in the EXAMPLES\OWL\TUTORIAL directory that are not discussed in this manual. These steps combine functionality from earlier steps into much more complex applications. They also extend the OLE ability of the

applications into OLE automated applications and automated controllers. Look through this code and see what you can learn from it.

You can also go through the examples in the other ObjectWindows example directories. Many of these have features in them you may want to try to add to the Drawing Pad application.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1245WW21777 • BOR 7775