OLE Programming with Borland C++ 5.0

Ted Faison
96/4/9

In the last few years Microsoft OLE has gone from being an esoteric buzzword to a widespread and dominant technology. OLE allows applications to be shared by other applications using a drag-and-drop approach. For years programmers have sought ways to leverage the power of applications like Word or Excel in their own applications. Custom controls only go so far. Being able to drop a full-fledged Word document into an application is a dream come true to users. Now they can run their familiar spelling checkers, add tables, format text in the genuine Word environment, without leaving the comfort of the enclosing application. But while users love OLE, programmers hate it. It has been said that the learning curve for OLE programming is comparable to that of Windows itself. To obviate much of the difficulty, Borland has added complete support for OLE development to its C++ compiler. To provide greater flexibility in the amount of OLE support provided, Borland has adopted a three-tier approach based on OWL, a separate framework called OCF, and a library called BOCOLE. Using Borland C++ 5.0 programmers can now develop all types of OLE objects: servers, containers, automated objects, automation controllers and OCX applications.

OLE is easy with Borland C++ 5.0

OLE support is built into OWL, but building an OLE application still requires knowledge of what OWL classes to use. To minimize the pain, AppExpert comes to the rescue, allowing a complete skeletal OLE object to be created automatically. Let's look at an example. You want to create a Windows 95-compliant app, meaning the app needs to be both an OLE server and container. But you also want to use OCX controls. No problem. Just pop open the AppExpert, select the **OLE 2 Options** page. You get the dialog box shown in Figure 1.

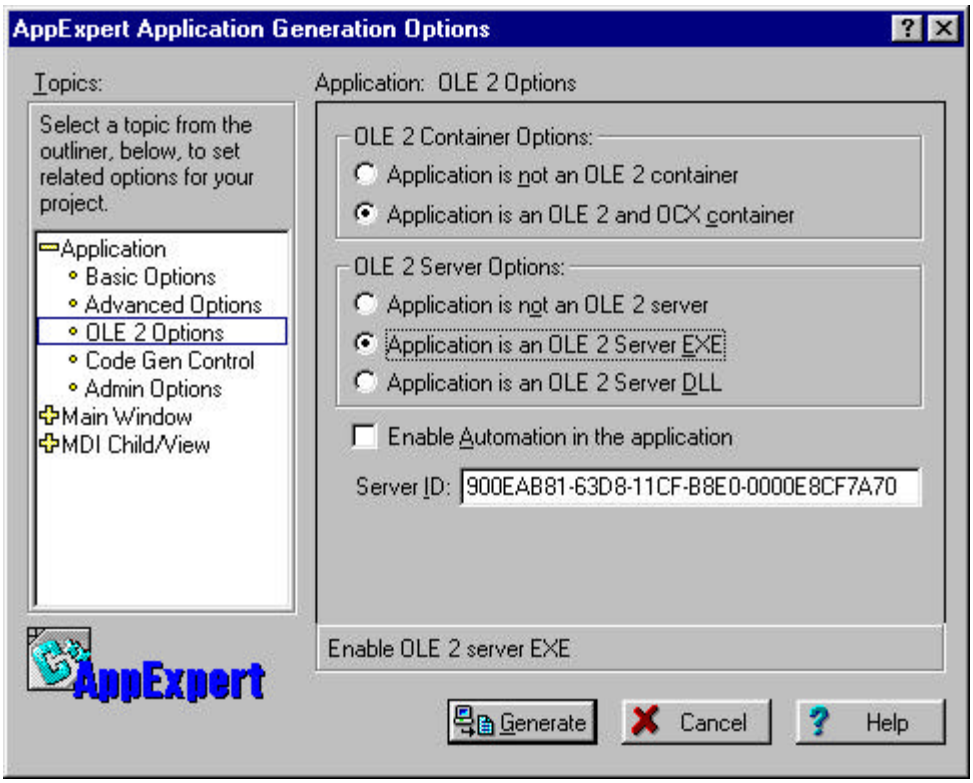

Figure 1 - The AppExpert Dialog Box showing the OLE options.

All you need to do is select the options shown in Figure 1 to enable the required OLE features. Click the **Generate** button and you're done. Running the sample app results in the following screen:

Figure 2 - The window displayed by the AppExpert generated sample program.

The Edit menu contains two important OLE-specific commands: **Insert Object** and **Insert Control**. The first brings up a dialog box showing you a list of all the OLE servers registered in your system. The dialog looks something like this:

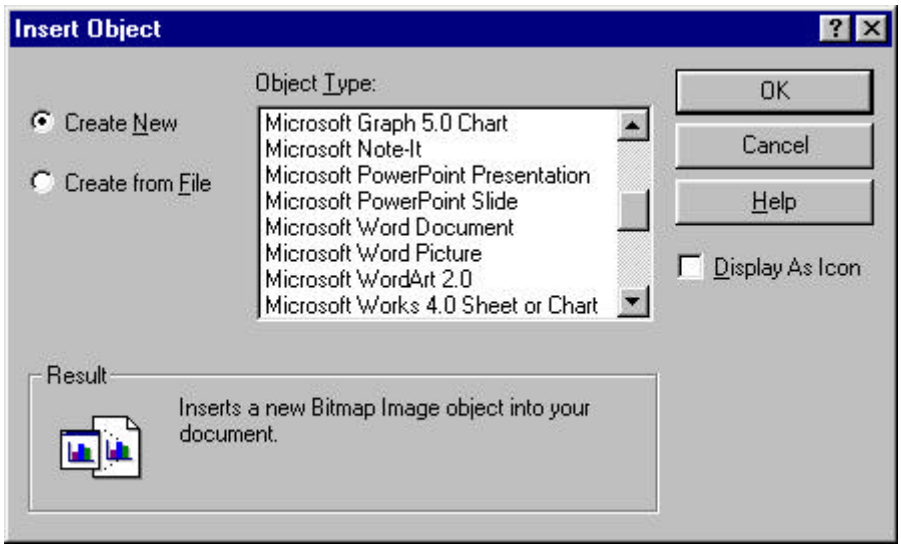

Figure 3 - The Insert Object OLE dialog box.

To insert an object into the sample app, just select it in the list and click **OK**. Figure 4 shows the screen after inserting a Microsoft Word document.

Figure 4 - A Microsoft Word document inserted into the AppExpert sample.

Adding OCX controls to the sample application is just as easy. The **Insert Control** command in the **Edit** menu displays a dialog box listing all the OCX controls registered in your system, as shown in Figure 5.

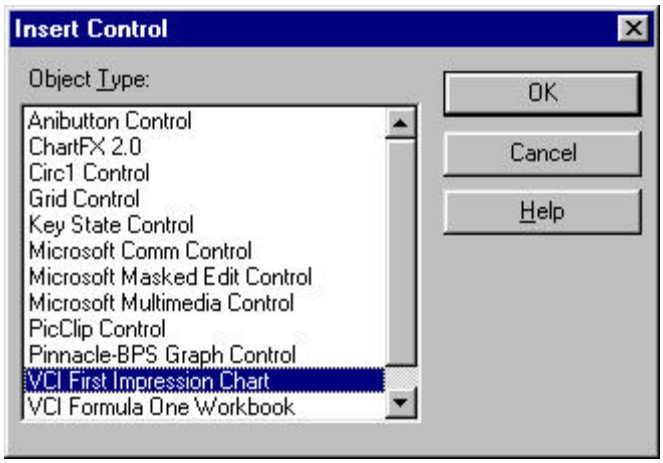

Figure 5 - The Insert Control OLE dialog box.

Choose the control you need and click the OK button. Figure 6 shows the sample application after inserting a 3-D graph OCX control. You'll almost always need to add some code of your own to manage OCX controls and handle notifications returned by them. A later section deals more with OCX programming issues.

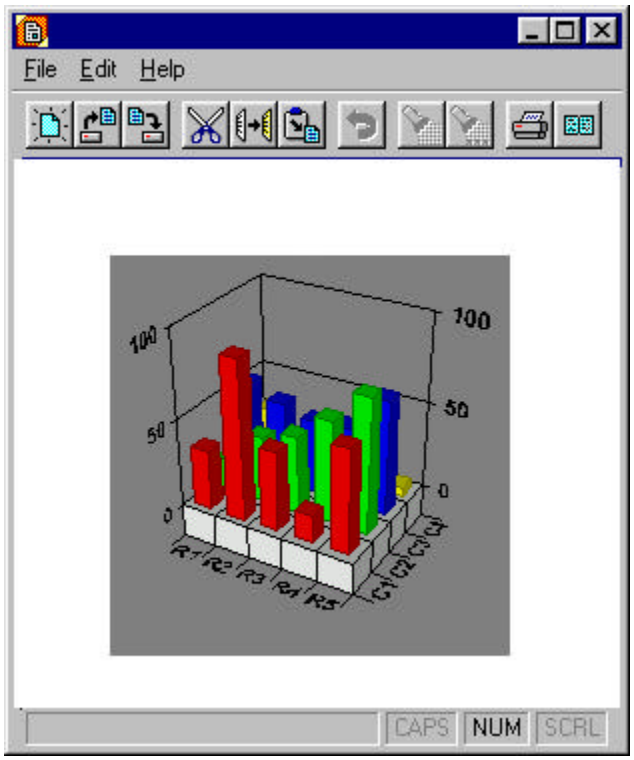

Figure 6 - An OCX graph control inserted into the sample app.

OLE objects are often referred to as *components*. The idea behind component-based programming is to utilize pre-packaged functionality as much as possible, leveraging the power of third party software that has already been tested and debugged. The quick examples shown emphasize the point-and-click approach to OLE. Borland C++ 5.0 makes it look easy because it

provides a great deal of code that silently works for you behind the scenes. The following section introduces some of the classes involved.

Under the Hood

When adding OLE support, the Borland developers had several options: add OLE-awareness directly to OWL, create a standalone class hierarchy to handle OLE, or use a layered approach. They chose the latter alternative, developing the 3-tier structure shown in Figure 7.

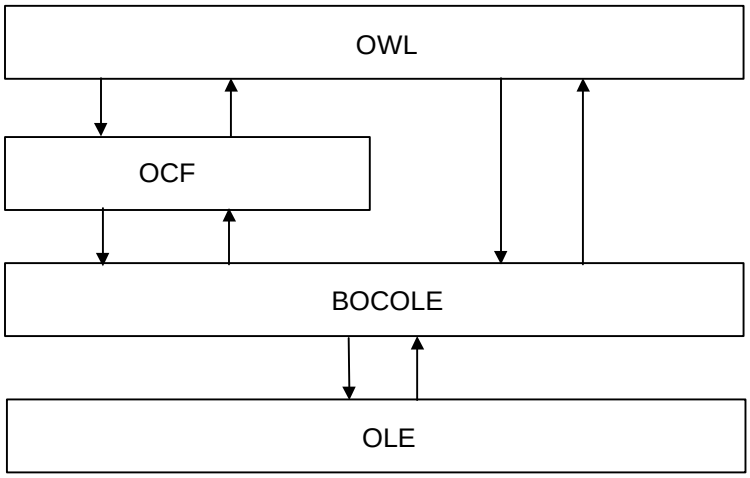

Figure 7 - The OLE programming model with Borland C++ 5.0.

Your application code runs on top of the OWL layer. OWL provides automated support for all the common OLE features, allowing programmers to development their application with little knowledge of OLE. OWL doesn't use low-level OLE calls, relying for the most part on services provided by a new class hierarchy called *ObjectComponents Framework* (OCF). OCF is a C++ package that provides a clean, object-oriented interface to the major OLE services, such as the following:

- Linking
- Embedding
- Automation
- Clipboard Operations
- Drag and Drop
- Compound Files

For many of the high-level OLE services there is a direct OCF class available. These classes are derived from the common OCF base class `TUnknown`, which supports the OLE interface `IUnknown`, so OCF objects are actually generic COM (Component Object Model) objects that can be used by applications written in any language that understands COM interfaces. To create an OLE-enabled application, your application must be derived from the mix-in class `TOcApp`. To embed an OLE server in an application, a `TOcPart` object is used to act as a proxy between the container and the server. The main OCF classes constitute a hierarchy, as shown in Figure 8.

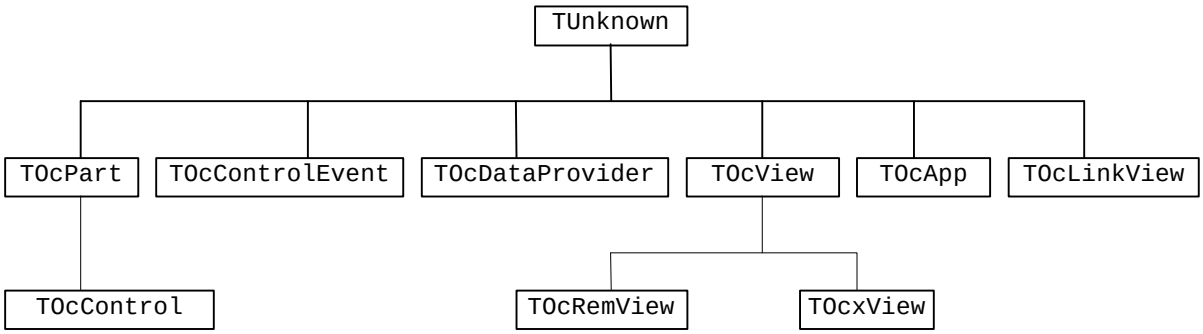

Figure 8 - The hierarchy of the main OCF classes.

There are several other OCF classes, including helper classes, non-COM classes and BOCOLE interface classes. The following table gives a brief description of the classes shown in Figure 8.

OCF Class	Description
`TUnknown`	Common base class, providing support for Microsoft COM architecture.
`TOcPart`	Acts as a proxy between OLE containers and embedded/linked objects.
`TOcControlEvent`	Handles events fired from OCX controls.
`TOcDataProvider`	Supports clipboard operations and drag-and-drop data transfers.
`TOcView`	Handles the display of embedded objects.
`TOcApp`	Base class necessary for apps needing OLE services.
`TOcLinkView`	Handles the display of linked objects.
`TOcControl`	Acts as a proxy between OCX containers and embedded OCX controls.
`TOcRemView`	Handles the display of server objects embedded in a container app.
`TOcxView`	Handles the display of OCX controls embedded in an OCX container.

Table 1 - The main OCF classes.

To create an OLE server or container, AppExpert creates a `TSDIDecFrame` main window derived from `TOleFrame`, which in turn uses an internal `TOcApp` object that it delegates OLE commands to. Using the Rumbaugh *Object Modeling Technique* (OMT), the class relationships are depicted in Figure 9.

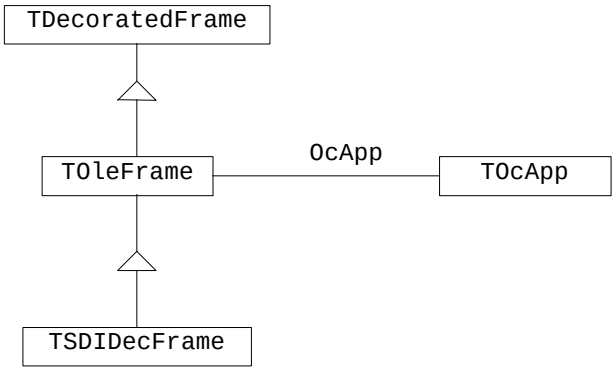

Figure 9 - An OMT diagram of the classes used to create the main window of an OWL OLE application.

The `OcApp` data member is created at the application level, in class `TOcAppHost`, from which `TSample1App` is multiply derived. The data member is shared with other classes, such as `TOleFrame`, which then uses it to handle certain OLE actions. For example, when `TOleFrame` receives a WM_SIZE message, it needs to notify linked or embedded objects inside it, so the message is passed along to `OcApp`. The WM_ACTIVATEAPP is handled the same way. `OcApp` also stores internally the OLE options, indicating whether the application is running as a server, an embedded object or a linked one. `TOleFrame` frequently calls `OcApp` to obtain the state of these OLE flags.

`TOleFrame` doesn't delegate all OLE functions to `OcApp`. In fact `TOleFrame` handles most OCF notifications itself. For example, when the user activates an OLE server embedded in a container app, OCF generates an OC_APPINSMENUS notification, requesting the server to merge its menus with the container's. `TOleFrame` handles the menu merging directly.

The view for the main window of an OWL OLE application is derived from a few standard OWL classes, as shown in Figure 10.

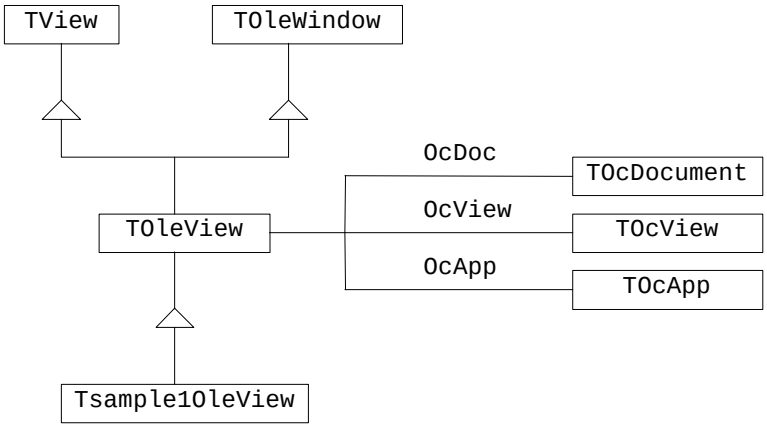

Figure 10 - The OMT diagram for the main view class of an OWL OLE application.

The `OcApp` object is the same used by `TOleFrame`. `OcDoc` is the OCF object that keeps track of objects linked or embedded in an OWL OLE container app. `OcView` handles the display of OWL servers linked or embedded inside OLE container apps. `OcApp` stores the OLE options for `TOleView`.

Automation is for developers

Although most of the attention in the media regarding OLE is on linking and embedding, for many developers the real power of OLE is in automation. Strictly speaking, automation is not even part of OLE proper, but an independent technology built on top of the COM architecture. Automation allows you to use third party applications as black box COM objects, vastly extending the notion of custom controls. For example Windows 95 has a rich text control that supports colored, bold and italic fonts. If you need more power, you have the option of inserting a complete Microsoft Word document into your app, and can control it via standard OLE automation commands.

To enable automation in an AppExpert app, you click the **Enable Automation in the application** checkbox in the window in Figure 1. If you create an app without automation

enabled, you can add it later with ClassExpert, by right-clicking a class in the **Classes** pane and selecting the **Automate Class** command, as shown in Figure 11. If the class is already automation-enabled, a **Delete Automation** command appears in the pop-up to let you remove automation at any time.

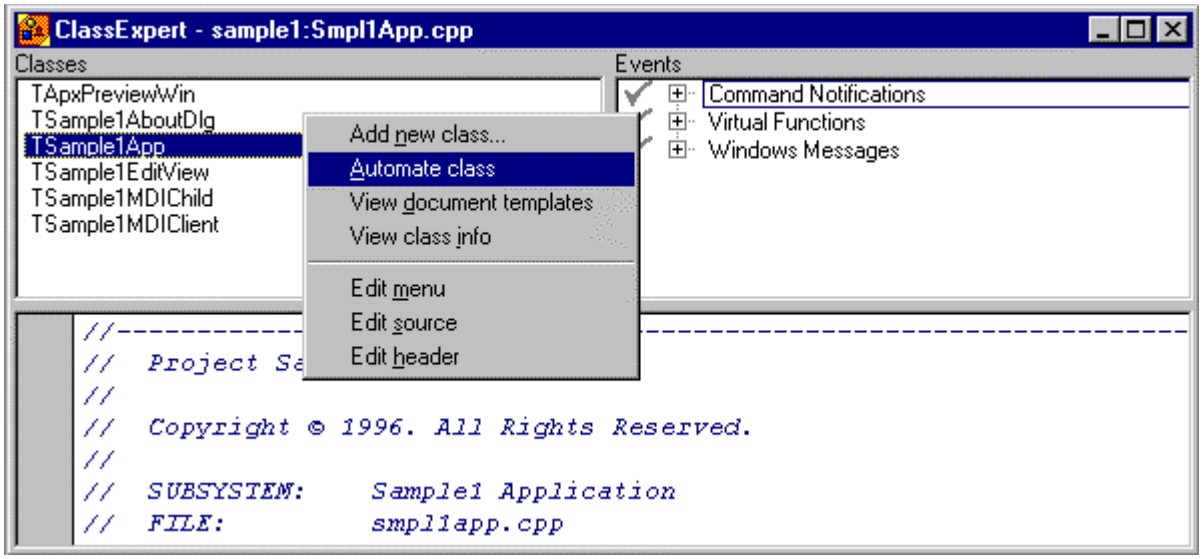

Figure 11 - Adding automation to a class using ClassExpert.

Enabling automation isn't enough for you to manipulate your application through automation, because an application will need to expose methods and properties for automation controllers to access. ClassExpert has been revamped to support the once-manual process of adding and exposing data, making it really easy.

Assume you have an app called `AutoServ`, which needs to expose a method called `Color`. Using the ClassExpert, you select the application class in the Classes pane, right click it and choose the **Automate class** command. Doing so adds the **Automation** entry in the **Events** pane, as shown in Figure 12.

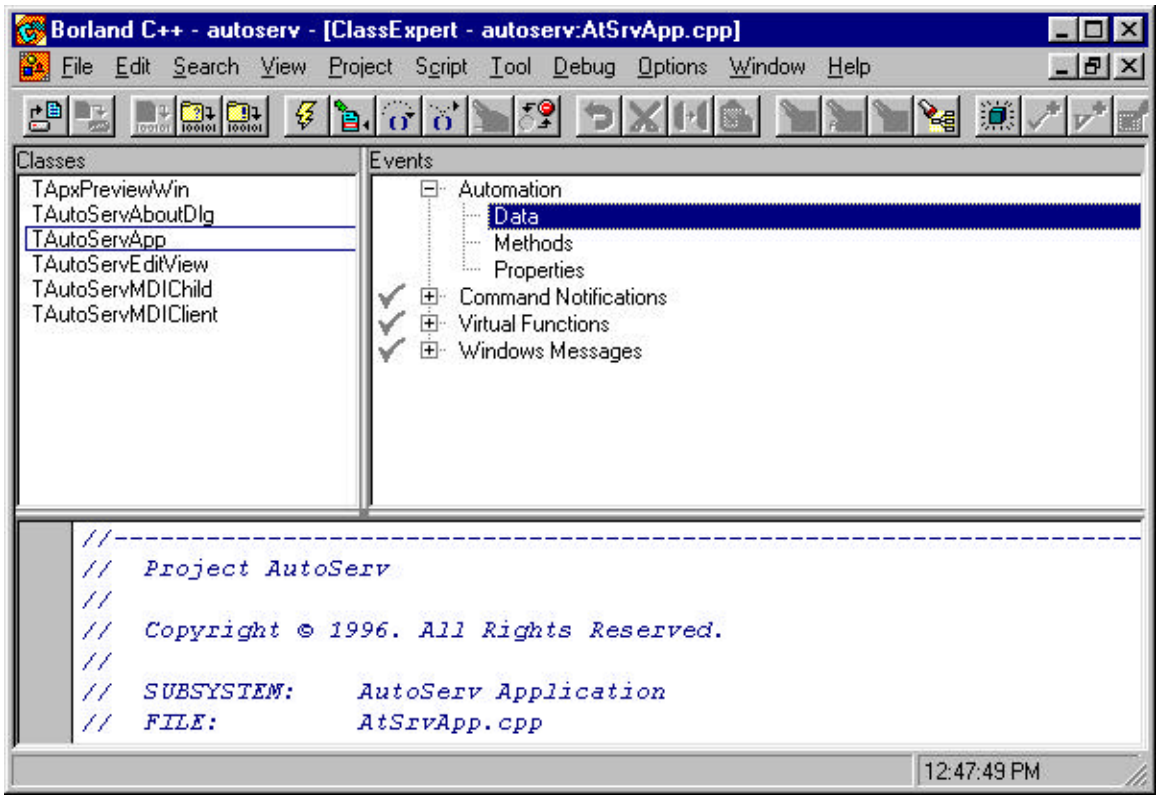

Figure 12 - The ClassExpert, showing the new support for Automation.

Right clicking on the **Data** field in the **Events** pane and selecting the **Add Data** command, you get a dialog box that allows you to specify the `Color` data member.

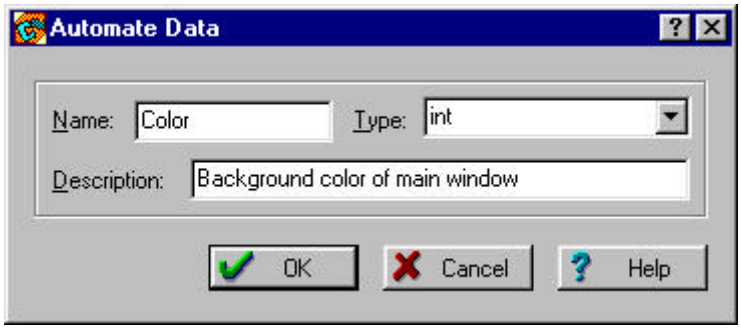

Figure 13 - The ClassExpert form that automates the creation of exposable automation data.

What ClassExpert does is add a public `int` data member to `TAutoServApp` called `Color`, like this:

```
class TAutoServApp : public TApplication {

  // … other declarations

  //{{TAutoServAppAUTO_DATA_BEGIN}}
  public:
  int Color;
```

```
  //{{TAutoServAppAUTO_DATA_END}}
};
```

ClassExpert also generates the macro

```
AUTODATA(Color, Color, int)
```

which exposes the member `Color` with the external name `Color` as an accessible integer value. ClassExpert has been extended to handle the creation of automation properties and methods. Properties allow controllers to get/set values, methods let controllers invoke functions inside automated objects.

OLE automation allows controllers to call server methods in two ways: using early binding and late binding. Early binding entails the use of a Type Library that describes the internal methods exposed by an automation server. Each method is associated with an integer called the *Dispatch ID*, or simply DISPID. With early binding, a developer compiles automation calls directly into the code, resulting in very fast code. The controller creates a proxy object for the server, then invokes methods directly off the proxy. For example, a controller using the `TAutoServApp` would invoke the `Color` method like this:

```
int color = server.GetColor();
```

where server is a proxy object bound to the `AutoServ` automated object. OCF adds code to your application to create a type library for you. To make this code run, you simply invoke the application using the `-TypeLib` flag and the type library is created. The library has the same name as the automation server, but has a `.OLB` suffix. The library is then added to the Windows registry for other applications to use. To generate the type library for `AutoServ`, you run the command:

```
AutoServ -TypeLib
```

Although OLE guidelines recommend that all automation servers register a type library, in practice many servers don't. For example Microsoft Word and Excel, which both can be used as automation servers, are not distributed with type libraries. With these types of applications, late binding is used. The OLE `IDispatch::GetIDsOfNames` function is called to obtain the DISPID of a function whose name is known. The DISPID is the ordinal number of a function in an interface, and is passed to the function `IDispatch::Invoke`. Due to the runtime lookup of the DISPID, using late binding is less efficient than early binding.

Component-based programming

One of the current trends in programming is develop objects as COM components, a technique that removes language dependencies from your objects. If you create a C++ object as DLL, the interface to the object uses exported names, which by default are mangled. Exported C++ functions also use parameter types that other languages may not understand, such as classes or pointers to classes. Creating a COM component out of a class forces the interface to adhere to the COM standard. Applications can use your component with much the same efficiency as any other C++ object, without encountering problems in exported names.

You can build COM objects with different levels of complexity with Borland C++ 5.0. The three basic types are

- A simple class object
- A library

- A complete application

To create a COM object out of a class, you add the OCF class `TUnknown` to its base class. You then override the `QueryInterface` function and process requests for interfaces you add to your class. By adding an `IDispatch` interface, applications can invoke functions in your COM object. A COM object this simple finds use as a way of subdividing large applications into smaller, standardized pieces.

To make COM objects that are entire DLLs or applications, the easiest way is to create the DLL or App using the AppExpert Options dialog shown in Figure 1, and click the **Enable Automation in the application** checkbox. You then use ClassExpert to automate specific classes in the application, as described earlier.

OCX control support

A new feature in Borland C++ 5.0 is support for OLE (OCX) controls. To talk to the control, there are two basic methods: the first uses late binding, and can be used with any OCX control. The second uses early binding, and can be used only with controls for which you have a type library.

To access a control using late binding, you use relatively low-level OLE functions. You start with the function `GetIDsOfNames` to obtain the dispatch ID of a function that you want to call. Using the dispatch ID, you can then call the `Invoke` function of the object's `IDispatch` interface. Assume you have an OCX control and you want to call its `Update` function. The following code could be used.

```
DISPID dispid;
HRESULT hResult = GetDispID(pDispatch, “Update”, dispid);
if (FAILED(hResult) ) return;

TVariantArg result(0);
DISPPARAMS dispParams = {0, 0, 0, 0};
UINT nErr = 0;
return pDispatch->Invoke(dispid, IID_NULL,
                         LOCALE_SYSTEM_DEFAULT,
                         DISPATCH_METHOD, &dispParams,
                         &result, 0, &nErr);
```

The `TVariantArg` object is used to pass parameters to functions invoked using `IDispatch::Invoke` calls. As you can see, using `GetDispId` and `Invoke` is a low-level approach. It also has poor run-time performance, but has the advantage of being useable with any OLE object, including those that aren't OCX controls. You can use the late binding approach, for example, to control an embedded Microsoft Word object or a third party OCX control for which you have no type library.

A better approach is the use of early binding, which is not only much higher level, but also faster at run-time. To use early binding, you run the Borland program AUTOGEN on your OCX control. AUTOGEN reads the control's type library and generates a C++ class that you can use as a proxy in your code. You can call your control's member functions directly through the proxy, eschewing OLE function calls completely. Assume you have an OCX control that has a member function called `Update`. AUTOGEN will generate a class that looks something like this:

```
class TMyControlProxy : public TAutoProxy {
public:
  TMyControlProxy() : TAutoProxy(DEFAULT_SYSTEM_LANGID) {}
  void Update();
  // …
};
```

To talk to the control, you simply instantiate an object of type `TMyControlProxy`, bind it to the OCX control, then use it directly. To call your control's `Update` function, all you have to do is this:

```
TMyControlProxy myControl;
myControl.Bind(pDispatch);
myControl.Update();
```

The parameter `pDispatch` is assumed to be a pointer to the OCX control's `IDispatch` interface. Using a proxy allows you to access properties and call methods using straight C++ code, so your code is not only shorter, but much easier to read and debug.

Handling Notifications

Dialog boxes that contain OCX controls must be derived from the OWL class `TOleDialog`. This base class provides the capability to handle OCX notification messages. When an OCX control fires a notification, it is caught by OCF, which then sends a WM_OCEVENT message with the notification code OC_CTRLCUSTOMEVENT. To add a generic handler for all OCX events, just override the function `EvOcCtrlCustomEvent` in your dialog box. This function is invoked with a pointer to a `TCtrlCustomEvent`, and receives complete information about the event and control that fired it. Your code might look something like this:

```
class TMyOCXDialog: public TOleDialog {

public:

  bool EvOcCtrlCustomEvent(TCtrlCustomEvent*);
}

bool TMyOcxDialog::EvOcCtrlCustomEvent(TCtrlCustomEvent* pev)
{
  if (pev->Ctrl == ptrSomeControl) {
    if (pev->Args->DispId == 4)
      // we received a notification of type 4
      // … do something
  return true;
}
```

The field `TCtrlCustomEvent.Ctrl` carries a pointer to the OCX control that fired the notification. To find the value of the notification, you use the `DispId` field in the `Args` field. The meaning of the notification code is entirely OCX control-dependent. Handlers return the value `true` to indicate they have processed an event.

Handling events through `EvOcCtrlCustomEvent` is a bit like handling Windows messages in `WndProc`. The preferred method is to provide a handler for the specific event you're interested in. The OCF library provides handlers for many common notifications, like mouse move and double-click events. Table 2 shows a list of these predefined handlers.

Event	OCF Handler to Override
Left mouse button click	`EvOcCtrlClick(TCtrlEvent*)`
Left mouse button double click	`EvOcCtrlDblClick(TCtrlEvent*)`
Left mouse button down	`EvOcCtrlMouseDown(TCtrlMouseEvent*)`
Mouse move	`EvOcCtrlMouseMove(TCtrlMouseEvent*)`
Left mouse button up	`EvOcCtrlMouseUp(TCtrlMouseEvent*)`
Key down	`EvOcCtrlKeyDown(TCtrlKeyEvent*)`
Key up	`EvOcCtrlKeyUp(TCtrlKeyEvent*)`
Error condition	`EvOcCtrlErrorEvent(TCtrlErrorEvent*)`
Change in focus status	`EvOcCtrlFocus(TCtrlFocusEvent*)`
Change in property	`EvOcCtrlPropertyChange(TCtrlPropertyEvent*)`
User wants to change a property	`EvOcCtrlPropertyRequestEdit(TCtrlPropertyEvent*)`
All events	`EvOcCtrlCustomEvent(TCtrlCustomEvent*)`

Table 2 - The OCF handlers for OCX notifications.

To handle a notification listed in Table 2, you simply override the default OCF handler. For example, to handle a mouse button down event, you would declare a handler like this:

```
class TMyOCXDialog: public TOleDialog {

public:

  virtual bool EvOcCtrlMouseDown(TCtrlMouseEvent*);
}
```

This way your function gets called on when the given control fires the given event, and you get passed pre-cracked parameters that correspond to notification you're getting. The `TCtrlMouseEvent` parameter carries the event information with it, and is declared like this:

```
struct TCtrlEvent {
  TOcControl* Ctrl;
};

struct TCtrlMouseEvent : public TCtrlEvent {
  short Button;
  short Shift;
  long  X;
  long  Y;
};
```

The mouse event handler would look something like this:

```
bool TMyOCXDialog::EvOcCtrlMouseDown(TCtrlMouseEvent* pev)
{
  char message [100];
  wpsrintf(message, "The mouse is at (%d,%d)", pev->X, pev->Y);
}
```

The OCX event-handling approach is similar to the method used by OWL to handle VBX notifications.

Conclusion

In order for your applications to obtain the Windows 95 logo, they must be both OLE servers and containers. Because of this, OLE has been promoted to a mainstream technology and will soon appear in thousands of new applications. To meet development schedules in today's competitive market, it is important to have solid tools that make OLE programming accessible to C++ developers.

Borland C++ 5.0 automates the process of setting up OLE applications using AppExpert. Adding customized functionality is easy using the new OWL classes and ClassExpert. Because most OLE functionality was built into the separate OCF class hierarchy, OWL retains its original structure and semantics. Programmers familiar with previous versions of OWL will continue to leverage their knowledge of the OWL framework, and can gradually familiarize themselves with OCF only when and if they need customized OLE functionality.