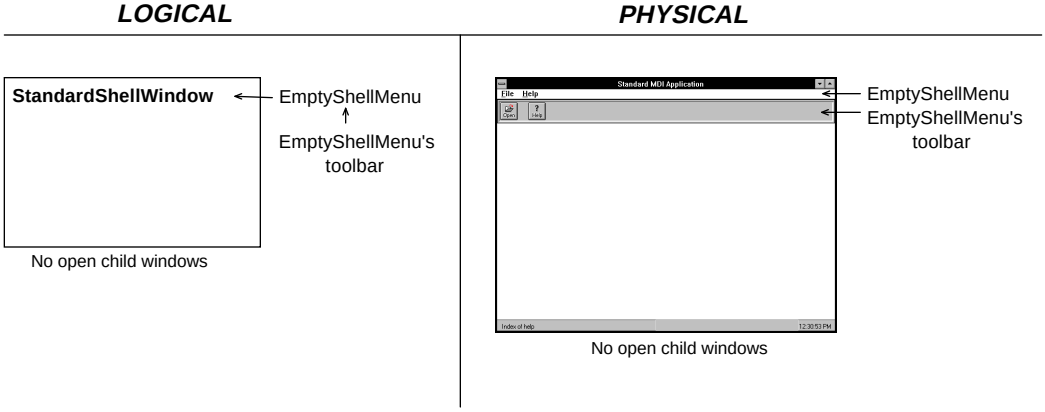
The Standard Shell Window

The relationships between the shell window, the EmptyShellMenu, and its toolbar are summarized in the diagram below:



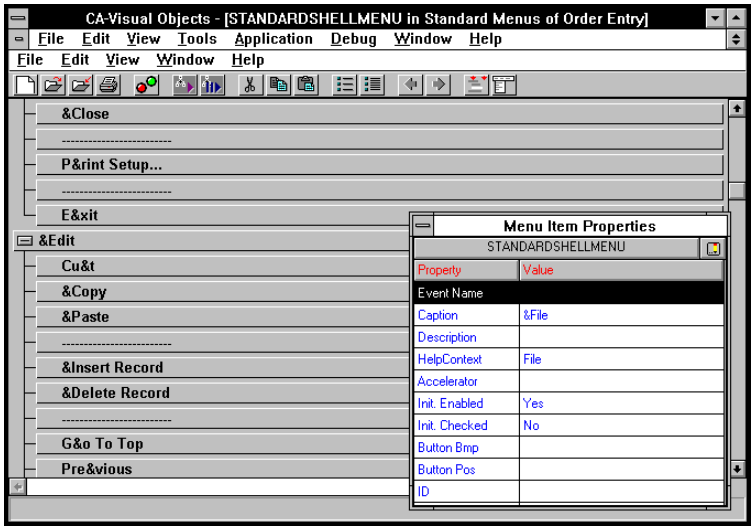
Opening a document (or child window) into the empty MDI shell window alters the nature of the shell window. It is no longer empty, but now holds another window in which data can be viewed and manipulated. When it contains one or more open documents, the shell window is referred to as the *standard* shell window. And, since the documents show data, more functionality is needed to view and manipulate the data—just the File and Help menus are no longer sufficient.

StandardShellMenu

In the Standard Menus Entity Browser, you will see another menu entity named StandardShellMenu, but you will need to scroll down to find it. Like EmptyShellMenu, this menu also has classes, resources, defines, and methods that help define it.

If you double-click on the StandardShellMenu menu entity, you can see that it contains not only File and Help menus, but several additional menus and commands that let the user manipulate the database in the child window.

For example, if you maximize StandardShellMenu, you can see standard Edit menu commands like Go to Top and Delete Record:



Also, like EmptyShellMenu, StandardShellMenu has its own toolbar. (When you are finished looking at the menu, close the Menu Editor by double-clicking on its system menu.)

Relationship to Child Windows

It is important to note that the shell window is not the owner of the StandardShellMenu. Instead, this menu is owned by the child window that is currently open and selected (or has “focus”). It is the child window that contains the data, and, therefore, it is the child window that requires the additional functionality provided by the StandardShellMenu. When a child window has focus, the shell window automatically “knows” to replace its own menu with the child window’s menu.

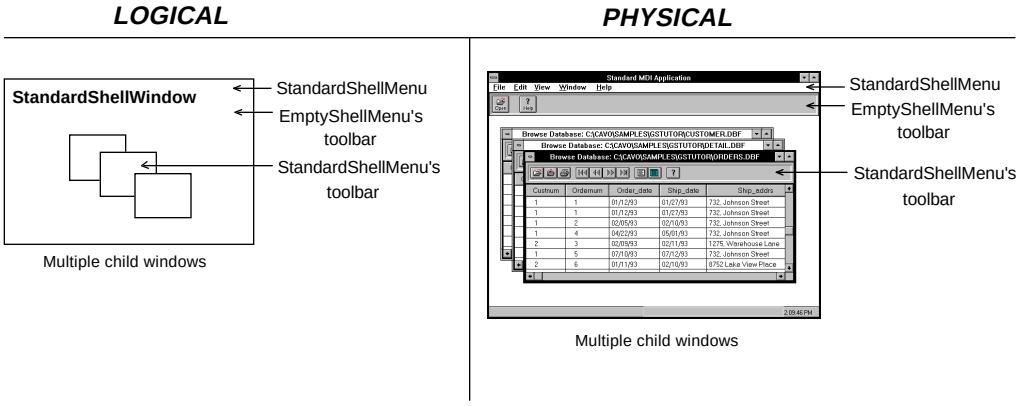
For example, you might develop a shell window into which the user can open a text editing window and a spreadsheet at the same time. In this situation, not only would the menu for each type of child window differ from that of the empty shell window, but the various types of child windows would most likely have menus that were different from one another. The shell window knows to display the appropriate menu, depending on which child window has focus.

The Standard Program takes care of this for you—when you open a child window for a database file into the shell window, the menu

displayed in the shell window is repainted, so that instead of EmptyShellMenu, the StandardShellMenu is displayed.

The replacement of one menu by another is accomplished through methods in the StandardShellWindow class. Basically, the menu selection to open a database file from EmptyShellMenu triggers an event that calls the StandardShellWindow:FileOpen() method which, in turn, calls a series of methods that instantiate the StdDataWindow class for the chosen database file. The Init() method of this class instantiates the StandardShellMenu class to display this new, more functional menu. (The StdDataWindow class is discussed in more detail in the next section.)

The relationships between the menus and toolbars displayed in the standard shell window and its child windows are summarized in the diagram below:



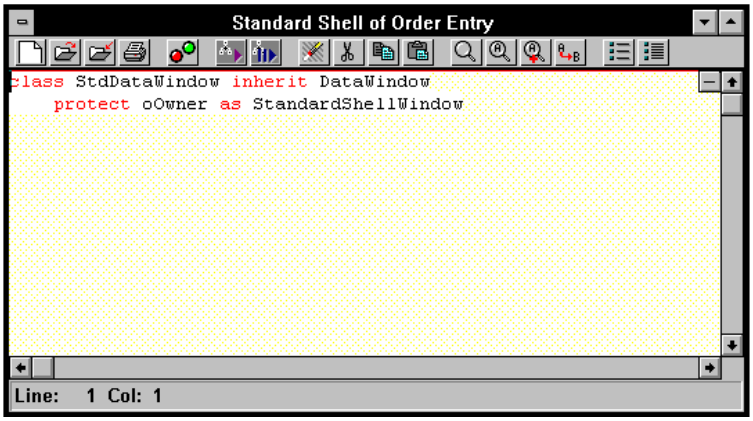
Note that while the shell window replaces its menu with that of the child window, it does not replace its toolbar. Instead, the shell window keeps its initial toolbar (EmptyShellMenu's), and each child window has its own toolbar (StandardShellMenu's).

The Child Windows

As mentioned in the previous section, the child windows opened in the shell window are instantiated (albeit indirectly) by the StandardShellWindow:FileOpen() method using a class named StdDataWindow that is defined in the Standard Shell module.

StdDataWindow

If you access the Entity Browser for Standard Shell (close the Standard Menus Entity Browser if it is still open) and double-click on the StdDataWindow class entity, you will see from the source code displayed that it derives from the DataWindow class:



Like the ShellWindow class, DataWindow is defined in the GUI Classes library.

Note: The second line of code defines a variable to identify the owner window.

Data Windows

A *data window* is a special kind of window that is capable of interacting intelligently with a database. The data window can easily display the contents of the database and has preprogrammed methods for moving among the records and manipulating the data (for example, Go to Top and Delete Record). In fact, the StandardShellMenu gets much of its functionality by directly invoking methods defined in the DataWindow class.

Data Servers

A *data server* is an object-oriented interface provided for interacting with a database. It is through the data server that the data window connects to a database and learns about its structure.

Although you can design both data windows and data servers for specific databases using editors built into the IDE (as you will see in subsequent lessons in this chapter), the Standard Program allows you to open *any* .DBF file or SQL data source by instantiating it as a data server (using the DBServer class defined in the DBF Classes library or the SQLSelect class defined in the SQL Classes library). It takes

advantage of the self-configuring properties of the DataWindow class to design a data window "on-the-fly" for that data server. It is the behavior that is built into the DBServer, SQLSelect, and DataWindow classes—rather than anything remarkable done by the Standard Program—that makes programming the self-configuring data windows so easy.

.DBF Files

In fact, the generated code is quite simple. For example, this is the flow of control for opening a .DBF file:

- StandardShellWindow:FileOpen() displays a standard File Open dialog box to allow the user to select a .DBF file name and calls its DoOpenFile() method using the resulting file name.
- DoOpenFile() verifies that it has a valid file name and calls NewDataWindow() using the file name.
- NewDataWindow() instantiates StdDataWindow using owner, file name, and data server parameters and registers this new window as a child of the shell window.
- Finally, StdDataWindow:Init() instantiates a generic data window, registers its owner and menu, instantiates a DBServer for the chosen file and links it to the data window, and displays the new data window in browse view.

Note: You can open all of these methods in the Source Code Editor at the same time and easily follow the logic described here. After you have loaded one (FileOpen(), for example), click the Open toolbar button to switch back to the Standard Shell Entity Browser, double-click on another entity (perhaps DoOpenFile()), and it will be added to the source code currently loaded in the Source Code Editor.

SQL Data Sources

The source code responsible for opening a SQL data source is located in the Standard SQL module. Among other things, this module contains code for a dialog window (StandardSQLDialog) that is opened in response to the File Open SQL menu command. Here is what happens when the StandardSQLDialog window is opened:

- StandardSQLDialog:Init() defines the controls for the dialog and StandardSQLDialogSub:Init() calls the GetTables() method.
- GetTables() displays the standard SQL Data Sources dialog box to allow the user to select a data source (this is accomplished by instantiating a SQLConnection object). Then, GetTables() populates the list box control on the StandardSQLDialog window using the tables in the selected data source.
- When control returns from GetTables() to Init(), the StandardSQLDialog window shows a list box of tables. When the user selects a table, the Open_Table() method is called.
- Open_Table() generates a SELECT statement based on the chosen table and instantiates a SQLSelect data server from the SQL data source using that statement.
- Finally, Open_Table() instantiates a StdDataWindow object for the server and changes the display mode of the new data window to form view.

Default Event and Error Handling

In a moment, we will build and run the Standard Program so that you can see the windows and menus in action, but first it will help to understand something about the basic event handling logic that is already built into the application. However, before moving on, close the Source Code Editor and the Entity Browser for the Standard Shell module. Doing so will return you to the Module Browser for the Order Entry application.

Event Handling

To briefly explain event handling, we'll start with the Windows environment. Windows provides a flexible, interactive environment in which multiple applications can be active and available simultaneously. To achieve this, all applications running under Windows interact with Windows—and possibly other GUI applications—through a *message queue*.

The message queue receives *messages* from both the operating system kernel and other applications. These messages are used to notify applications of *events* that require attention. For example, if the user presses a button or selects a menu in an application, an event is posted to the message queue. (The Windows message queue maintains both user-generated and system-generated events.)

The message queue then notifies the various applications of the events that pertain to them. When programming for Windows, therefore, your applications must know how to *handle* (interpret, respond to, and generate) events.

In CA-Visual Objects, the basic event handling logic is this: a command event (such as a menu or push button selection) is sent first to the lowest-level window that has focus. If that window has no mechanism for dealing with the event, the event is passed up to the window's owner. This propagation of an event up the ownership chain continues until some window handles the event or until it finally reaches the App, where it most likely ends up doing nothing.

Some examples of built-in event generation and handling in the Standard Program have already been mentioned in this section. For example, in the empty shell window when the File Open menu command or the Open toolbar button is selected, a FileOpen event is generated, causing the StandardShellWindow:FileOpen() method to be invoked.

The event names for menu commands are defined in the Menu Editor, as you will see later in this chapter. CA-Visual Objects processes these events automatically by trying to match the event name first to a method, then to a Window or ReportQueue subclass of the same name. The event is then handled either by invoking the method or instantiating an object of the subclass.

This automatic propagation is quite useful. In an MDI application, for example, File Save and File Print are normally handled by the child window because these commands are specific to each document, while File Open and File Print Setup are more general and are, therefore, normally handled by the shell window.

Again, you can see this illustrated in the Standard Program. The StandardShellMenu has both a File Open menu command and an Open toolbar button, but the child window that owns this menu does not have a FileOpen() method. So, when the FileOpen event is generated from a child window, it ends up being handled by the StandardShellWindow:FileOpen() method.

Error Handling

There is also some error handling provided by the Standard Program. For example, if you attempt to open anything other than a .DBF file with the File Open menu command, you'll receive a message similar to the following: