

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain



[Comment on this article](#)



It is possible to use the Visual C++ development environment as a big C compiler. Using it that way you can certainly create GUI applications using the Win32 API. However, you will be wasting a lot of the potential of the environment. Visual C++ contains a variety of visual tools that, once you know how to use them, can tremendously accelerate your software development cycle. These tools, which include the AppWizard, ClassWizard and the resource editors, make full use of the C++ language and the MFC (Microsoft Foundation Classes) class hierarchy.

The problem with these tools is that they are fairly complicated, and there is not much introductory material available on how to use them. The purpose of these tutorials is to show you, using extremely clear and easy-to-understand examples, exactly what the AppWizard and ClassWizard are, what they can do, and exactly how to use them. Along the way you will find out why the AppWizard generates 20 files and what all of those files do, what the "document/view paradigm" is and how it makes application development easier, and how to fit the code you develop into an AppWizard framework with a minimum amount of effort on your part. Once you have finished these tutorials you will have a very clear understanding of why these tools exist and how you can use them to create your own applications.

Note: If you do not know C++ these tutorials will be opaque to you. Please run through the [C++](#) tutorials first. If you do not have any knowledge of MFC these tutorials will be fairly muddy. Please look at the introductory [MFC](#) tutorials first.

Here is a list of topics discussed in these tutorials:

- [Introduction to the AppWizard, ClassWizard and resource editors](#)
- [The Document/View Paradigm](#)
- [Understanding the AppWizard files](#)
- [Creating a drawing editor](#)
- [Synchronizing Views](#)
- [Understanding Document Templates](#)

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? devcentral@iticentral.com

[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain

Introduction to the AppWizard, ClassWizard and the Resource Editors.

The Visual C++ development environment comes with three different tools that can tremendously accelerate your software development cycle: The AppWizard, the ClassWizard and the resource editors. The latter is a collection of editors that you use individually to edit specific resource types. This tutorial will introduce you to these tools in a general way.

The *AppWizard* is a "template generator." Think about what you normally do when you begin any new project. Typically you would find some piece of code that you know works, and then you would strip out all of the things that you do not need from the old application. Then you would add in all of the functionality for the new application. The AppWizard saves you the first step. It generates a clean code template for you to use as the starting point for any new application that you want to create. The AppWizard gives you a number of different options as you are creating the template, and you choose among these options depending on what type of application you are planning to create. You will use the AppWizard exactly once for each application you create. After it has generated the template, the AppWizard is finished.

The template, or framework, that the AppWizard generates is an extremely complete MFC application consisting of twenty or so different files. See tutorial 3 for a description of these different files.

The *ClassWizard* is a tool that you can use to perform specific modifications to an AppWizard framework. In particular, you will most often use the ClassWizard to modify message maps, to generate new classes derived from existing MFC classes, and to add member variables to certain classes. The ClassWizard can also perform several other tasks related to OLE automation code and OCX code. The ClassWizard is only useful if you are working within a framework generated by the AppWizard.

Resources are user interface objects that you can create with visual *resource editors*. Resources include bitmaps, cursors, dialogs, icons, menus and their accelerator table, string tables and version information. Although you could do what resources do with normal MFC code (for example, you could hand-code a menu or a dialog using normal MFC code written in C++), resources are generally much easier to create and apply in a program. Therefore, the use of resources speeds your application development cycle. Resources have the added advantage of concentrating language-specific parts of your user interface in a specific place that is completely separate from your C++ code. So, for example, you can create an English, French and Spanish version of your resources and, without changing any of your C++ code, easily create three different versions of your application in three different languages.

When you are creating applications using these tools, you will typically start by creating the application's initial framework with the AppWizard. The framework will include all of the normal menu options like File Open, Edit Cut/Copy/Paste, and the Help menu. You will then use the Resource Editors to add new menu items, dialogs, etc. to your application. You will use the Class Wizard to modify message maps, override virtual functions, add new classes, and so on to the application. Because these tools do so much of the grunt work for you, they significantly increase your development speed and reduce the number of errors you make.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? devcentral@iticentral.com

[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain

Understanding the Document/View Paradigm

The framework that the AppWizard generates revolves around a concept called the *Document/View Paradigm*. If you understand this paradigm then it is much easier to understand the files that the AppWizard generates, and it also makes it much easier for you to fit your code into the AppWizard framework. This tutorial describes the paradigm so that you completely understand its purpose and intent.

The App Wizard takes a document-centric approach to application design. The MFC class hierarchy contains two classes that help to support this approach: **CDocument** and **CView**. The AppWizard and MFC use this approach because most Windows applications work this way. Built into any framework generated by the AppWizard is the assumption that your application will want to load and work with multiple documents, and that each document will have one or more views open at a time. This approach makes it extremely easy to create both Single Document Interface(SDI) and Multiple Document Interface(MDI) applications. All applications can be thought of in terms of documents and views.

It is easiest to understand the document/view paradigm if you think about a typical MDI word processor like Microsoft Word. At any given time you can have one or more documents open. A document represents a single open file. The user generally has one view open on each document. The view shows the user a part of the document in an MDI window, and lets the user edit the document. However, Microsoft Word allows the user to split a window into multiple frames so that the user can have two or more views on the same document if desired. When the user edits in one of the views, it changes the data in the document associated with the view. If a document has multiple views open and the user changes data in one of the views, the document and all other related views should reflect the change. When the user saves the document, it is that data held by the document that gets saved to disk.

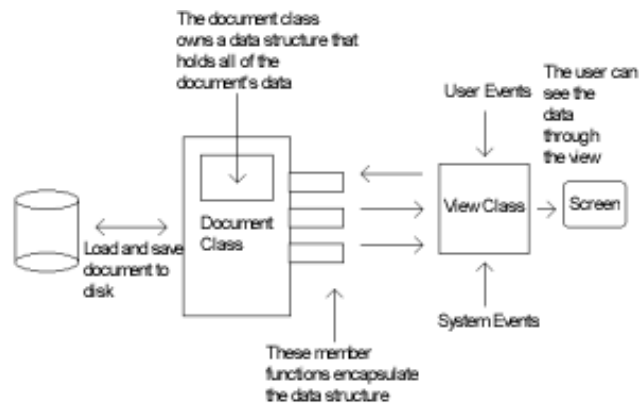
Many applications allow the user to open just one type of document. Microsoft Word, for example, works only with Microsoft Word documents. It may allow you to open other types of documents, but it first filters them to turn them into Word documents. Other applications open several different types of documents and can display all of them simultaneously in its MDI framework. Visual C++ is an example of this type of application. The most common type of document Visual C++ works with is a text file that contains code. However, you can open resources in the different resource editors as different types of documents in the MDI framework. Microsoft Works is similar. It can open word processing documents, but it can also open spreadsheet and database documents. Each of these documents has a completely unique view in the MDI frame, but all of the different views live there in harmony with one another. In addition, database documents in Works can be viewed both in a spreadsheet-like list, or in a customizable form that shows one complete record at a time.

Therefore, in the most general case, an application may be able to open several different types of documents simultaneously. Each type of document can have a unique viewing window. Any document may have multiple views open at once. In fact, a document might have more than one way of viewing its data, and those different views on a single document might be open simultaneously. Each document stores its data on disk. The views give the user a way to view and edit that data.

At a code level, the document and view classes separate functionality. Since this arrangement is typical of most applications, the framework generated by the AppWizard supports this structure implicitly. The document class is responsible for data. It reads the data from a file on disk and holds it in memory. The view class is responsible for presentation. The view takes data from the document class and presents it to the user in a view. The multiple views for a single document synchronize themselves through the data in the document. The MFC class hierarchy contains classes - **CDocument** and **CView** - that make this structure easy to create. The AppWizard derives new classes from the existing document and view classes and you build your application within those derived classes.

The goal of the document class is to completely encapsulate the data for one open document. It holds the data for the document in a data structure in memory and knows how to load the data from the disk and save it to the disk. The view class uses and manipulates the data in the document based on user events. The view class is responsible for letting the user view the contents of the document. (Note - The document class does not necessarily have to hold its data in memory. In certain types of applications it can act instead as a pipe to a binary file that remains on disk, or to a database.)

The relationship between the document and view classes is summarized in the figure below. When you are designing your own applications, you want the document class to completely encapsulate the data, and you want the view class to display information to the user. There should be a clear and obvious way for the view to interact with the document through member functions that you create if you want to properly encapsulate the data in the document.



When you create your own application, you typically will create some sort of data structure in the document class to hold the document's data. MFC contains a number of collection classes that you can use for this purpose, or you can create any other data structure that you like. The document class will be called when the data in the data structure needs to be loaded from disk or saved to disk. This is most commonly done through the document class's **Serialize** function. You will then add your own member functions to the class to encapsulate the data structure and allow the view class to manipulate the data held by the data structure. The view class contains the code to handle user events and draw to the screen.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 *Interface Technologies, Inc.* All Rights Reserved
Questions or Comments? devcentral@iticentral.com
[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by **Marshall Brain**

Introduction to the AppWizard Files

The framework that the AppWizard generates typically consists of at least 20 different files. The first time that you see those files and begin to wade through them they can be extremely confusing unless you have an appropriate roadmap. The purpose of this tutorial is to provide you with that roadmap.

Let's start by creating a simple framework with the AppWizard. To create this framework we will use all of the default settings for the AppWizard's options. Once the AppWizard has generated the files for this default framework we can go through each one to see what they all do.

Select the **New** option in the **File** menu. In the dialog that appears, make sure that the **Project** tab is selected. We want to create an AppWizard workspace, so select "MFC AppWizard(exe)" from the choices. Along the right side of the window is a dialog that lets you name the project, choose the project type, and pick the project's directory. Choose the appropriate directory in which to create the new samp directory by leaving it blank (a new directory will be created in your current directory) or type the path directly. Name the project "samp". This name will be used as the new directory name as well, and that is fine - you will see the word "samp" echoed in the Location field as you type it. Click the **OK** button to create the new project.

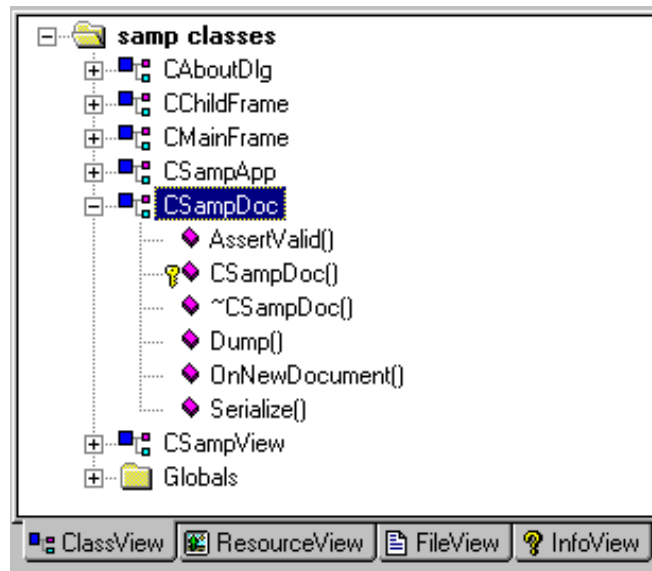
You will next see a group of six colorful option dialogs. You can move between them with the **Next** and **Previous** buttons. Look through them now. We do not want to change any of the default options, so when you are through looking at them click the **Finish** button. You will see a final dialog that summarizes your choices. Click **OK** on this final dialog and the AppWizard will create the new framework, after displaying what options have been selected so that you can make sure it is what you really want.

Use the File Manager to look into the new directory that the AppWizard created. It will contain about 20 files. Here's a quick summary of all of these different files:

- SAMP.DSP, SAMP.DSW, SAMP.NCB, SAMP.OPT - The project and workspace files for the application.
- SAMP.CLW - A data file used by the ClassWizard that you will never touch. If you accidentally delete it the ClassWizard will regenerate it for you.
- README.TXT - A text file that briefly describes most of the files in the directory.
- STDAFX.H, STDAFX.CPP - Short little files that help in the creation of *precompiled header* files. To speed compilation, Visual C++ can parse all of the header files (which are gigantic) and then save an image of the compiler's symbol table to disk. It is much quicker for the compiler to subsequently load the precompiled header file when compiling a CPP file. You will never touch these files.
- SAMP.H, SAMP.CPP - If you look inside these files you will find that they contain a class derived from **CWinApp**, as must all MFC programs. There is also an overridden **InitInstance** function. You will sometimes have occasion to modify these files, and will use the message map in the application class to handle messages that apply application-wide.
- MAINFRM.H, MAINFRM.CPP - These files contain a class derived from **CMDIFrameWnd**. In an SDI application the class is derived from **CFrameWnd** (you control whether the AppWizard generates an SDI or MDI application in the first of the AppWizard's six configuration screens). You will typically leave these two files alone.
- CHILDFRM.H, CHILDFRM.CPP - These files contain a class derived from **CMDIChildWnd** and control the look of the child windows in the MDI shell. In an SDI application these files are omitted. You will typically leave these two files alone.
- SAMPDOC.H, SAMPDOC.CPP - These two files derive a new class from **CDocument**. You will modify these files to create the document class for your application.
- SAMPVIEW.H, SAMPVIEW.CPP - These two files derive a new class from **CView**. You will modify these files to create the view class for your application.
- RESOURCE.H, SAMP.RC, SAMP.APS - These three files compose the *resource file* for your application. You will find in a moment that the file contains an accelerator table, an About dialog, two icons, two menus, a fairly extensive string table, a toolbar, and version information.

Typically you will modify only the last seven files when creating a new application. You will use the ClassWizard to help with the modifications to the document and view classes, and you will use the resource editors to modify the resource files.

A quick note on the workspace view in Visual C++ 6.x. The workspace view typically appears on the left side of the Visual C++ windows and looks like this (if you cannot see it, choose the **Workspace** option in the **View** menu):



The project window has 4 tabs along the bottom. In the figure the **ClassView** tab is selected. This view shows you all of the classes in the application. If you click on the small plus signs, you can see the functions in each class. You can double click on a class or function name and VC++ will take you to that point in the code. The **ResourceView** tab shows you the resources in the application. Open all of the resources and look at them by double clicking. You will find that this application has an accelerator table, an about dialog, two icons, two menus, a string table, a toolbar and some version information. The **FileView** tab shows you the files that make up the application (if you are upgrading from VC++ 2.x this is the view you are familiar with). You can delete files from the project here by selecting a file and hitting the delete key. The **InfoView** tab shows you the help files.

Build and execute this framework so that you can see its default behavior. To do this, choose the **Build samp.exe** option in the **Build** menu, and then choose the **Execute samp.exe** option in the **Build** menu. You will find that the application has the expected menu bar, and that several of the menu options fully or partially work. For example, the **Help** option brings up an about dialog and the **Open** option brings up an open dialog. The application has a toolbar, a status bar, an about box, etc. In other words, it is a fairly complete application framework.

To get an idea of how you might modify this framework using the ClassWizard, try the following example. Select the **ClassWizard** option in the **View** menu. Make sure that the **Message Maps** tab is selected. Make sure that the **CSampView** class is selected in the **Class Name** combo box. In the **Object Ids** list choose the first item: **CSampView**. In the **Messages** list choose WM_MOUSEMOVE. Click the **Add Function** button. Make sure that the **OnMouseMove** function is selected in the **Member Functions** list and click the **Edit Code** button. The ClassWizard will create the new **OnMouseMove** function, add it appropriately to the message map, and then deposit you at that point in the view class. Modify the function so that it looks like this:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+5, point.y+5);
    }
    CView::OnMouseMove(nFlags, point);
}
```

Build and execute the application. When you drag the mouse in the window you will find that it draws small 5 by 5 rectangles wherever the mouse goes. The **CClientDC** class creates a "device context" that allows you to draw in a window. We use the dc to draw each rectangle. **CClientDC** derives its behavior from the **CDC** class, which you will find, when you look it up in the MFC help file (click on the word and then press F1), contains about 100 different drawing functions. Experiment if you like with these functions. They are all fairly self explanatory. While you are in the help file it wouldn't hurt to take a look at the **CDocument** and **CView** classes to start getting a feel for them.

To get an idea of how the resource editors work, open SAMP.RC by clicking on the ResourceView tab in the project window. Open the **Dialog** folder and then double click on the IDD_ABOUTBOX dialog and the template for the About box will appear. Click on one of the static text strings to bring it into focus. Hit **Enter** and a dialog appears which you can use to modify the string. Under the **General** tab, change the text in the "Caption:" field. Build and execute to see the changes to the About box.

In the next tutorial we will make a number of changes to the document class, view class and resource file to create a simple but complete drawing editor.

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? devcentral@itcentral.com

[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain

Creating a Simple Drawing Editor

Probably the best way to understand the document and view classes, as well as the ClassWizard and resource editors, is to run through a simple example application that makes use of all of these features in straightforward ways. In this tutorial we will use the AppWizard framework that you generated in the previous tutorial. As you recall, we modified it so that you could draw small rectangles using the mouse. However, it has no way to load or save the "drawings", nor can it refresh the screen on exposure. In this tutorial we will solve these problems using the Document/View paradigm. We will also add a new menu option and dialog to the application to demonstrate simple uses of the resource editors.

Start with the "samp" application framework that you created in tutorial 3. In order to turn this framework into a complete application we must take the following steps:

- Add a data structure to the document class to hold the points that the user draws.
- Add code to the document class so that the data structure in the document class is properly loaded from and saved to disk.
- Add code to the view class so that whenever the user draws a point it is added to the document's data structure.
- Add code to the view class so that it properly redraws the drawing when the application receives an exposure event.

Altogether, we will have to write only about 15 lines of code to accomplish all of this.

The following instructions walk you through these different steps.

Step 1: Add a data structure to the document class

MFC contains a variety of different collection classes, and by using them we can very easily add a robust data structure to hold the points that the user draws. Look up the collection classes in the MFC help file by choosing the **Search** option in the **Help** menu, and then typing "hierarchy chart". There should be a category toward the right of the chart with Arrays, Lists and Maps. These are the collection classes.

In this application we need to store the points drawn by the user. This is most easily done using a pair of **CDWordArrays**. These are simple array classes that store values of type **DWORD** (a DWORD being a 32-bit signed integer), and we will store the x value of each point in one of the arrays and the y value of each point in the other. Click on the CDwordArray class in the hierarchy chart to learn more about the class. The MFC array classes have a number of useful features that will make our lives easier: The MFC arrays automatically size themselves as new elements are added and give you virtually infinite room, they do range checking in debug mode, and they know how to read and write themselves to disk.

Open the SAMPDOC.H file (probably the easiest way to do this is to double click on the CSampDoc class in the ClassView seen in tutorial 3, or go to the FileView and double click on SAMPDOC.H in the Header Files folder) and find the public attributes section. There will be a section labeled "//attributes" with a "public:" marker within it. In that position type:

```
CDWordArray x, y;
```

If you then save the header file, the ClassView of the project window will display the two new variables in the CSampDoc class.

As an alternative, you can use the tools. Right-click on CSampDoc in the ClassView, and then choose **Add Member Variable...** from the list. You will have to do this twice, once for x and once for y. Both variables will be added to the public implementation section of the header file, and you can watch it happen if you have the header file open. For now, put x and y in the attributes section.

In this simple example we will treat the arrays as public members to make things more obvious, but in a real application it would be beneficial to make the data structure private and provide new member functions in the document class to allow the manipulation of the data structure. These functions would allow for the complete encapsulation of the document class.

Step 2: Allow for loading and saving of the document's data structure

The AppWizard framework and the MFC classes do a good bit of the work related to loading and saving files to disk. For example, when you choose the **Open** option of the **File** menu in your drawing program, it already pulls up the proper File Open dialog. The framework then takes the file name selected by the user, opens it, creates a binary "archive" and attaches it to that file, calls several functions in the **CDocument** class, and finally calls a function named **Serialize** in the CSampDoc class, passing it the archive (to see all of this code in action, finish adding the code described in this tutorial and put a break point in the document's **Serialize** function, run the program under the debugger, choose File Open, and then, when the program stops, choose the **Call Stack** option in the **View** menu. You will be able to examine the MFC source code). This same **Serialize** function is also called when it is time to save the file. All that you have to do is fill in this function and your data structure will automatically be loaded and saved to disk. To make matters even easier, the **CDWordArray** class (and all other MFC collection classes) know how to serialize themselves.

Find the **Serialize** function in the CSampDoc class. The easiest way to do this is to open the ClassView, click on plus sign next to CSampDoc, and then double click on the Serialize function. Change it so that it looks like this:

```
void CSampDoc::Serialize(CArchive& ar)
{
    x.Serialize(ar);
    y.Serialize(ar);
}
```

The **x** and **y** variables will handle all details of serialization, including understanding whether they should load or save themselves to the archive. The archive knows which direction the data should move.

Step 3: Modify the view class so it saves points into the document's data structure

First we will add a member variable to the view class to store the size of the rectangles that the editor draws, rather than using the hard-coded value of 5 as in tutorial 3. We will then later add a dialog that lets us change this value in order to demonstrate the addition of dialogs. Open SAMPVIEW.H and find the public attributes section. Add the following declaration:

```
int w;
```

Note that in this same section the view has a member function named **GetDocument** that returns a pointer back to the view's document. This function will be important in a moment.

Now find the constructor for the CSampView class (you can double-click on it in the ClassView and add the following line to initialize the new member:

```
w = 5;
```

In the CSampView class, find the **OnMouseMove** function that you added in tutorial 3 (you can find it by hand in SAMPVIEW.CPP, or use the ClassView). Change the function so that it looks like this:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+w, point.y+w);
        pDoc->x.Add(point.x);
        pDoc->y.Add(point.y);
        pDoc->SetModifiedFlag();
    }
    CView::OnMouseMove(nFlags, point);
}
```

Look up **OnMouseMove** in the MFC help file for information on its parameters. The easiest way to do this is to select Search in the Help menu and type OnMouseMove.

The function has been modified so that it retrieves a pointer to the view's document and then saves the current point in the document's **x** and **y** members. The function also calls the document's **SetModifiedFlag** function (see the MFC help file) to set the document's dirty bit. Once set, the document will automatically query the user about saving the file if the user attempts to close it or quit without saving. This function also makes use of the new **w** member in the view.

Step 4: Handle exposure events in the view class

Whenever the view class receives an exposure (WM_PAINT) event, it calls the **OnDraw** member function. This function also is called to handle printing. By putting exposure-handling code into this function you can complete the application. Find the **OnDraw** function in the CSampView class. Modify it as shown below:

```
void CSampView::OnDraw(CDC* pDC)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int i;
    for (i=0; i<pDoc->x.GetSize(); i++)
        pDC->Rectangle(pDoc->x[i],
            pDoc->y[i],
            pDoc->x[i]+w,
            pDoc->y[i]+w);
}
```

This function simply asks one of the arrays how many elements it contains and then runs through a **for** loop that many times, drawing all of the points that the user has previously entered.

Compile and run the application. You will find that it properly handles exposure events, and that it can now load and save files to disk. You have properly modified the document and view classes to create a complete, fully functional drawing editor. Congratulations!

Note that, as described in tutorial 3, the document class holds the data structure and loads and saves the data to disk. The view class uses the data in the document to handle redrawing, and manipulates the data when the user moves the mouse. This is proper use of the document and view classes.

It would not be a bad idea to now stop, close this tutorial, and try to recreate the application "blind". You will find that you generate a number of questions when you try to do it without help, and those questions can teach you a lot.

What we would like to do now is add a new menu option and dialog to the application so that the user can modify the size of the rectangles that the application draws. This exercise will show you how easy it is to modify the menu and create new dialog resources.

Step 5: Add a new menu option

In the application's workspace window choose the ResourceView tab and double click on the Menu resource section. You will find that the application has two menus. Double click on both and look at them. IDR_MAINFRAME is short, and is used when there are no windows open in the MDI framework. IDR_SAMPSTYPE is longer and appears when windows are open. We want to modify IDR_SAMPSTYPE, so open it by double clicking on it.

At the end of the menu bar you will find an empty rectangle. Click on it and type "&Option". Press the Return key. The & character indicates which letter in the menu title you want to use as a mnemonic, and will appear underlined when the menu is shown on the screen. You can move the & anywhere within the string. Now a new rectangle will appear below **Option**. Click on it and type "&Size". Press the Return key. Now click on **Option** and drag it to a more appropriate place in the menu bar, perhaps between **Edit** and **View**. That's all it takes to create a new menu and option.

Now double-click on the new size option. Note that the editor has automatically assigned it the obvious ID of ID_OPTION_SIZE. You can change the ID as you see fit, but there is rarely a reason to do so. We will use this value inside a message map to respond to the new option. [note: If you are unfamiliar with the concept of a message map, visit the [MFC Tutorials page](#) and see the introductory MFC tutorial]

Build and execute the application. You will find that it has a new menu option but that the option is disabled.

Now open the ClassWizard by choosing the **ClassWizard** option in the **View** menu. Make sure the **Message Maps** tab is visible. Make sure the **CSampView** class is selected. In the **Object IDs** list choose ID_OPTION_SIZE. In the **Messages** list choose COMMAND (UPDATE_COMMAND_UI is used primarily to enable and disable the menu options - look up **CCmdUI** in the MFC help file for more information). Click the **Add Function** button. The ClassWizard will pick the obvious name **OnOptionsSize** for the new function and show it to you. Click **OK** to accept the name. Click the **Edit Code** button with the **OnOptionsSize** function selected and modify the function as shown below:

```
void CSampView::OnOptionsSize()
{
    Beep(500, 500);
}
```

Build and execute the application. Now when you choose the **Size** option you will find that the application beeps. You can see that it is trivial to wire new menu options into an AppWizard application.

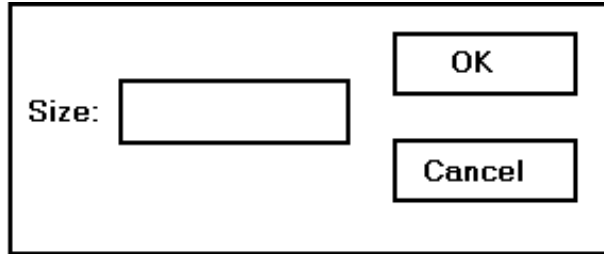
Step 6: Add a new dialog

To add a new dialog to the application we have to do four things:

- Create a new dialog template with the dialog resource editor,
- Create a new dialog class for that template,
- Add DDX variables(s) for the controls in the dialog, and
- wire the dialog into the application.

Here are the steps:

Select the **Resource...** option from the **Insert** menu. Choose **Dialog** and click on **New**. You should see a new dialog containing OK and Cancel buttons, and a small palette of tools should also appear. If the palette is not in evidence, choose the **Customize** option in the **Tools** menu and check the box marked **Controls**. One button on the palette says "Aa" and is used to create static text controls. Another is labeled "ab|" and is used to create editable text controls. Create a dialog that looks like this:



Click on the new edit control and press **Enter**. You will find its ID to be IDC_EDIT1. This is fine here, but in a real dialog you would likely change it to something more meaningful. Select the OK button and note its ID is IDOK. This is normal and you will not want to change it. Right click in the title bar of the new dialog itself and choose the **Properties** option. Note that the dialog's ID is IDD_DIALOG1. Note also that you can change its title here.

With the dialog still open on the screen, choose the **ClassWizard** option in the **View** menu. The ClassWizard will see the new dialog and assume that your desire is to create a new dialog class for it. This class will act as a liaison between the dialog resource and your application, and you will need to create a new dialog class for each dialog that you add to an application (although you will rarely or never touch this class except through the ClassWizard). The first dialog you see lets you specify that you want to create a new dialog class. There are several fields in the new dialog class creation dialog. In the **Name** field type "CSizeDlg". Make sure that **Base Class** contains **CDialog**, that **File** contains SIZEDLG.CPP, that the **Dialog ID** field contains the dialog's ID of IDD_DIALOG1, and that **OLE Automation** is set to None. Click the **OK** button to create the dialog class. Click the **OK** button in the ClassWizard to close it.

To try out the new dialog, Find the **OnOptionsSize** function in the CSampView class. Change it so it appears as below:

```
void CSampView::OnOptionsSize()
{
    CSizeDlg dlg;
    dlg.DoModal();
}
```

Also, be sure to include SIZEDLG.H as the last header file included in SAMPVIEW.CPP.

Build and execute the application and select the size option. You will find that the dialog appears properly and disappears as expected when you click the OK or Cancel buttons.

What we would like to do now is get the value typed by the user into the edit field so that we can modify the view's **w** member. Select the **ClassWizard** option in the **View** menu. Select the **Member Variables** tab at the top of the ClassWizard. Make sure that **Class Name** is set to **CSizeDlg**. We want to add a member variable to the **CSizeDlg** class to allow us to get the value from the dialog's edit control. In the list, double click on IDC_EDIT1. In the dialog that appears, set the **Member Variable Name** to "m_size". Set the **Category** to "Value". Set the **Variable Type** to UINT. Click the OK button. In the new **Minimum Value** and **Maximum Value** fields that appear at the bottom of the ClassWizard type 2 and 50 respectively.

The **m_size** variable is a **DDX** variable. DDX=Dialog Data Exchange. This new variable will always contain the value that the user types into the edit control, or you can set it to display a default value to the user. The values you typed for the minimum and maximum are known as **DDV** values. DDV=Dialog Data Validation. Anything the user types will be checked against these values when the user clicks the dialog's OK button.

Replace the code in **OnOptionsSize** with the following:

```
void CSampView::OnOptionsSize()
{
    CSizeDlg dlg;
    dlg.m_size = w;
    if (dlg.DoModal() == IDOK)
        w = dlg.m_size;
}
```

Build and execute the program. You will find that if you change the value in the dialog, the size of the rectangles drawn by the application will change appropriately. The code is simply setting or retrieving the value from the edit control using the **m_size** variable as its proxy. The value in **m_size** is copied into the edit control when the dialog appears, and the value in the edit control is copied into **m_size** when the user clicks the OK button.

You may notice that the edit control does not initially have focus. Fix this by opening the dialog resource and choosing the **Tab Order** option in the **Layout** menu. Click on each control in order to establish the tab order.

You may want to store the **w** value with each point that the user draws. To do this, add a new **CDWordArray** variable to the document class, serialize it appropriately, and change the view class to set and retrieve this array's values in the same way you change **x** and **y**.

Conclusion

In this tutorial you have seen how easy it can be to create and modify a very robust and capable program using the AppWizard, ClassWizard and the resource editors. In the next tutorial we will fix one niggling little problem left in this application.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved
Questions or Comments? devcentral@iticentral.com
[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain

Synchronizing Views

In the previous tutorial you learned how to modify the document and view classes to create a simple drawing editor. There is one subtle problem with that program, however. In this tutorial you will learn how to solve that problem using the view's **OnUpdate** function.

To demonstrate the problem, run the application that you created in the previous tutorial. Draw something in the default window. Now choose the **New Window** option in the **Window** menu. This option opens a second view on the same document. This second window will display the same thing that the first window does because both share the same document. Now choose the **Tile** option in the **Window** menu. You can see that both views are identical. Now draw into one of the views. What you will find is that the views will not be synchronized. What you draw into one view does not appear in the other. However, if you iconify the application and then expand the icon, you will find that the views are once again identical. Both receive exposure events, and both draw from the same document data, so they must look the same.

What we would like to do is modify the code so that, when you draw in one view, all views attached to the same document are immediately updated as well. The framework already contains the functions necessary to do this—all you have to do is wire them in properly.

The **CDocument** class maintains a list of all views attached to the document. It also contains a function called **UpdateAllViews**. This function, when called, calls the **OnUpdate** function of each view attached to the document. By default the **OnUpdate** function does nothing, but you can modify it to do anything you like. Optionally you can pass the **OnUpdate** function two programmer-defined parameters to further customize its activities.

What we would like to create here is a mechanism that causes all views attached to a document to paint the last point added to the data structure whenever any of the views for that document adds a new point. To do this, first modify the **OnMouseMove** function in the view class so that it contains a call to **UpdateAllViews**, as shown below:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+w, point.y+w);
        pDoc->x.Add(point.x);
        pDoc->y.Add(point.y);
        pDoc->SetModifiedFlag();
        pDoc->UpdateAllViews(this, 0, 0);
    }
    CView::OnMouseMove(nFlags, point);
}
```

This call to **UpdateAllViews** indicates that the document should call the **OnUpdate** function in all views attached to it *except* the one indicated by **this**. It does this because the current view has already drawn the point and there is no reason to do it a second time. The latter two parameters in the call to **UpdateAllViews** will be passed directly to **OnUpdate**. We do not have any use for these parameters in this simple example so we pass zeros. It would not hurt to read about both **CDocument::UpdateAllViews** and **CView::OnUpdate** in the MFC help file. Also read about **CView::OnInitialUpdate** while you are there.

Now use the ClassWizard to override the **OnUpdate** function. Choose the **ClassWizard** option in the **View** menu. Make sure that the **Message Maps** tab is selected. Make sure that **CSampView** is the class selected in the Class Name field. Click on **CSampView** in the **Object IDs** list. Search down until you find **OnUpdate** in the **Messages** list. This function is a virtual function and we can override it with the ClassWizard. Select **OnUpdate** in the list, click the **Add Function** button and then click the **Edit Code** button. Modify the function so that it looks like this:

```
void CSampView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int i = pDoc->x.GetSize();
    if (i > 0)
    {
        i--;
        CClientDC dc(this);
        dc.Rectangle(pDoc->x[i],
                    pDoc->y[i],
                    pDoc->x[i]+w,
                    pDoc->y[i]+w);
    }
}
```

The goal of this function is to get the last point in the data structure and draw it. It therefore gets the size of one of the arrays, checks to make sure that the array is not empty, and then draws the last point. The **if** statement is necessary because the **OnInitialUpdate** function gets called when the view is created, and by default it calls **OnUpdate**. You could override this function to remove the default behavior and the **if** statement would no longer be necessary. However, it is not a bad safety feature.

Build and execute the application. Choose the **New Window** option in the **Window** menu, followed by the **Tile** option. Draw in one of the windows and you will find that both views update simultaneously. This is proper behavior, and will work regardless of the number of views that are open on the same document. It is also very efficient.

There are other ways in which to use the **UpdateAllViews/OnUpdate** to accomplish the same thing. For example, **OnMouseMove** might draw nothing and let the **OnUpdate** function handle all drawing. Or you might pass the new point as one of the parameters. Experiment with different techniques until you find the one you like best.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

by Marshall Brain

Understanding Document Templates

One of the more interesting, and best hidden, features of any AppWizard framework is something called *document templates*. In this tutorial you will learn about document templates and see how you can add new ones to your applications. By the end of the tutorial you will have used document templates to create a single MDI application that can display both text and drawing documents simultaneously.

Creating a Text Editor

Let's start by using the AppWizard to create a second type of application. In the previous tutorials we have created a drawing editor. Here we will quickly create a text editor. It is interesting to note that you can create a complete text editor - one with all the features of Notepad, along with several others as well - without writing a single line of code. Take the following steps:

- In Visual C++, select the **New** option from the **File** menu, make sure the **Project** tab in the subsequent dialog is selected, and name the new project "Ed". Make sure the type is set to **MFC AppWizard (EXE)** and select an appropriate directory. Click **OK** and look over the AppWizard's options in the six configuration screens.
- We want to change two things in the configuration screens: First we want to give a default file extension, and second we want to change the view class. In the fourth screen of the six, click the **Advanced** button and type "tex" into the **File Extension** field. In the sixth screen, click on **CEditView**, and at the bottom of the dialog change the **Base Class** to **CEditView** using the combo box.
- Compile and run the program. You will find that you have a complete MDI text editor. You can load and save text files, cut, copy and paste text, print files, and so on.
- If you look at the help page for the **CEditView** class, you will find that it automatically understands certain menu options. In particular, if you add menu options with the IDs of **ID_EDIT_FIND**, **ID_EDIT_REPEAT**, **ID_EDIT_REPLACE** and **ID_EDIT_SELECT_ALL**, the program will *automatically* recognize these new options and perform them properly. You do not need to add anything but the menu options. Do that now and test the program.

This application was so easy to create because the **CEditView** class has all of the behavior of a normal text editor built into it. There is just one line of code that the AppWizard had to add to make the whole thing work, and that line can be found in the **Serialize** function of the document class. It looks like this:

```
((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
```

That line loads and saves text files. Just so you are aware of it, the **CEditView** class violates the strict separation of document and view. The **CEditView** class contains a normal **CEdit** control, and this control holds the editor's data itself. Therefore, the data resides inside the **CEditView** class rather than in the document class, and the line of code above gets or sets that data. Because of this odd structure, you will want to remove the **New Window** option from the **Window** menu. Since the document does not hold the data, it is not possible to have multiple views display the same document. This seems like a small price to pay for the ease of using the **CEditView** class to create quickie text editors.

Now that you have created a complete text editor, let's see what steps are necessary to create a single MDI application that can display both text and drawing documents. To do this, we will take the drawing program from the previous tutorials and modify it so that it can also display text documents. To do this, three steps are required:

- Step 1: Start with the drawing program and add a new document and view class for the text editor
- Step 2: Create a new document template for the new document type
- Step 3: Add three new resources to the drawing editor

Once you have completed these three steps, the program will be able to display both text and drawing documents simultaneously.

Step 1: Add a new document and view class

Open the workspace file for the drawing editor (samp) in Visual C++. Choose the **ClassWizard** option in the **View** menu. Click the **Add Class** button and select **New**. You will see a dialog with several fields. In the **Name** field type **CEdDoc**. In the **Base Class** field choose **CDocument**. The ClassWizard will choose a file name of EDDOC.CPP, and this name is fine. Click the **OK** button. Click the **Add Class** button again to create another new class. Type **CEdView** into the **Name** field and choose **CEditView** for the base class type. The file name EDVIEW.CPP chosen by the ClassWizard is fine. Click the **OK** button. Close the ClassWizard by clicking its **OK** button.

Now modify the **Serialize** function in the new document class (CEdDoc) so it contains the line seen in the text editor:

```
((CEditView*)m_viewList.GetHead()->SerializeRaw(ar);
```

The two new CPP files were automatically added to the drawing project by the ClassWizard. You will add the header files to the CSampApp class in the next step.

Step 2: Add a new document template

Open the CSampApp class, which contains the application class for the application derived from **CWinApp**, in the ClassView so that you can see a list of its functions. Find the **InitInstance** function. Double click on it. Look for the following lines:

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_SAMPSTYPE,
    RUNTIME_CLASS(CSampDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CSampView));
AddDocTemplate(pDocTemplate);
```

These lines create a *document template*. The **CWinApp** class (see the help file) has built into it the ability to hold a list of document templates. When it holds more than one, it changes the behavior of the application slightly. For example, the **New** option pops up a list that lets the user choose what type of document he/she wishes to create. What we want to do is change the program so that it contains two templates: one for drawing documents, and another for text documents. Modify the above code so that it looks like this:

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_EDTYPE,
    RUNTIME_CLASS(CEdDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CEdView));
AddDocTemplate(pDocTemplate);
pDocTemplate = new CMultiDocTemplate(
    IDR_SAMPSTYPE,
    RUNTIME_CLASS(CSampDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CSampView));
AddDocTemplate(pDocTemplate);
```

Note that a new document template has been created. The new one goes first (more on the reason for that in a moment). It specifies **IDR_EDTYPE**, **CEdDoc** and **CEdView**. But what does that mean?

The purpose of a document template is to relate a resource type (**IDR_EDTYPE**), a document class, a view class, and a window class. When the application framework needs to create a new instance of a document for the user, it looks to the document template, which tells it to create a new instance of the appropriate document, view and window classes. The resource ID is used when the framework needs to change resources. It identifies a specific menu, icon and string resource. So, for example, when the user clicks on a window in the MDI shell, the application framework brings that window to the foreground *and* it changes the menu to the one appropriate for that window, according to the window's document template.

We put the text document template first because, if the user attempts to open a document whose extension is unknown to the application, the application tries to open it under the first document template registered. Since text documents are far more likely than drawing documents, the text template is placed first in the list of document templates.

Be sure to include EDDOC.H and EDVIEW.H at the top of SAMP.CPP.

Step 3: Create resources

The new document template specifies a resource ID of **IDR_EDTYPE**. If you open the ResourceView and look through its resources, you will find that it already contains three resources of type **IDR_SAMPSTYPE** as needed by the drawing editor: a menu, an icon, and a string near the top of the string table. The easiest way to create new resources for the text editor type is to copy these three **IDR_SAMPSTYPE** resources to the clipboard, paste them back, and then change their names to **IDR_EDTYPE** using the **Properties** option in the **View** menu. Then modify them as appropriate. For example, to the **IDR_EDTYPE** menu you will want to add the **ID_EDIT_FIND**, **ID_EDIT_REPEAT**, **ID_EDIT_REPLACE** and

ID_EDIT_SELECT_ALL options (also delete the **Option** menu that got copied). You will also want to remove the **New Window** option from the Window menu. Change the IDR_EDTYPE icon as you see fit. Change the IDR_EDTYPE string so that it looks like this:

```
\nEd\nEd\nEd Files (*.tex)\n.TEX\nEd.Document\nEd Document
```

For more information about this mysterious string, search for the **GetDocString** function in the help file. It will explain what all seven of the substrings do. Now that you understand the strings, modify the IDR_SAMPTYPE string as well so it contains a file extension:

```
\nDrawing\nDrawing\nDrawing Files (*.drw)\n.DRW\nDrawing.Document\nDraw Document
```

Change the two strings in any way that you like.

Step 4 : Build and execute

Build the new application and run it. When it starts you should see a new dialog that lets you choose whether you want to create a text or drawing document. Choose text, and verify that the text editor works properly. Now choose **New** and create a drawing document. Draw something. Note that when you change windows the menu bar changes as appropriate.

Conclusion

You can see that adding a new document template is easy, and there is really no limit to the number of templates a single application might have. As you create more complex applications, you will find this to be an extremely useful feature of the AppWizard framework.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved
Questions or Comments? devcentral@iticentral.com
[PRIVACY POLICY](#)